YAZ User's Guide and Reference

Sebastian Hammer

Adam Dickmeiss

Edited by

Adam Dickmeiss

YAZ User's Guide and Reference

by Sebastian Hammer

by Adam Dickmeiss

Edited by Adam Dickmeiss

Copyright © 1995-2003 by Index Data

This document is the programmer's guide and reference to the YAZ package version 2.0. YAZ is a compact toolkit that provides access to the Z39.50 and SRW protocols, as well as a set of higher-level tools for implementing the server and client roles, respectively. The documentation can be used on its own, or as a reference when looking at the example applications provided with the package.

Table of Contents

1. Introduction **1**
- Reading this Manual 1
- The API 2

2. Compilation and Installation **5**
- Introduction 5
- UNIX 5
  - Compiling from source on Unix 6
  - How to make apps using YAZ on UNIX 8
- WIN32 9
  - Compiling from Source on WIN32 9
  - How to make apps using YAZ on WIN32 11

3. ZOOM **13**
- Connections 13
  - Z39.50 Protocol behavior 15
  - SRW Protocol behavior 15
- Queries 15
  - Protocol behavior 16
- Result sets 16
  - Z39.50 Protocol behavior 17
  - SRW Protocol behavior 18
- Records 18
  - Z39.50 Protocol behavior 19
  - SRW Protocol behavior 20
- Scan 20
- Options 21
- Events 21

4. Generic server **22**
- Introduction 22
- The Database Frontend 22
- The Backend API 23
- Your main() Routine 23
- The Backend Functions 25
  - Init 25
  - Search and retrieve 27
  - Delete 29
  - scan 30
- Application Invocation 30

5. The YAZ client **34**
- Introduction 34
- Invoking the YAZ client 34
- Commands 35
- Searching 38

6. The Z39.50 ASN.1 Module 40
Introduction 40
Preparing PDUs 40
Object Identifiers 41
EXTERNAL Data 42
PDU Contents Table 44
7. SOAP and SRW 50
Introduction 50
HTTP 50
SOAP Packages 50
SRW 52
8. Supporting Tools 55
Query Syntax Parsers 55
Prefix Query Format 55
Common Command Language 58
CCL Syntax 58
CCL Qualifiers 60
CCL API 60
CQL 61
CQL parsing 62
CQL tree 62
CQL to PQF conversion 64
Specification of CQL to RPN mapping 65
CQL to XCQL conversion 67
Object Identifiers 67
Nibble Memory 70
9. The ODR Module 72
Introduction 72
Using ODR 72
ODR Streams 72
Memory Management 72
Encoding and Decoding Data 73
Diagnostics 76
Summary and Synopsis 77
Programming with ODR 77
The Primitive ASN.1 Types 77
INTEGER 78
BOOLEAN 78
REAL 78
NULL 78
OCTET STRING 79
BIT STRING 79
OBJECT IDENTIFIER 80
Tagging Primitive Types 80
Constructed Types 80
Tagging Constructed Types 82

Implicit Tagging 82
Explicit Tagging 82
SEQUENCE OF 83
CHOICE Types 84
Debugging 87
10. The COMSTACK Module 88
Synopsis (blocking mode) 88
Introduction 89
Common Functions 89
Managing Endpoints 89
Data Exchange 90
Client Side 91
Server Side 92
Addresses 93
Diagnostics 93
Summary and Synopsis 94
11. Future Directions 96
A. License 97
Index Data Copyright 97
Additional Copyright Statements 97
B. About Index Data 99
C. Credits 100

List of Tables

3-1. ZOOM Connection Options 14
3-2. ZOOM Result set Options 16
3-3. ZOOM Scan Set Options 20
6-1. Default settings for PDU Initialize Request 44
6-2. Default settings for PDU Initialize Response 44
6-3. Default settings for PDU Search Request 45
6-4. Default settings for PDU Search Response 45
6-5. Default settings for PDU Present Request 45
6-6. Default settings for PDU Present Response 46
6-7. Default settings for Delete Result Set Request 46
6-8. Default settings for Delete Result Set Response 46
6-9. Default settings for Scan Request 47
6-10. Default settings for Scan Response 47
6-11. Default settings for Trigger Resource Control Request 47
6-12. Default settings for Resource Control Request 47
6-13. Default settings for Resource Control Response 48
6-14. Default settings for Access Control Request 48
6-15. Default settings for Access Control Response 48
6-16. Default settings for Segment 48
6-17. Default settings for Close 49
9-1. ODR Error codes 76

List of Figures

1-1. YAZ layers 2

List of Examples

8-1. PQF queries 57
8-2. Small CQL to RPN mapping file 66

Chapter 1. Introduction

YAZ is a C/C++ library for information retrieval applications using the Z39.50/SRW protocols for information retrieval.

Properties of YAZ:

- Complete Z39.50 (http://www.loc.gov/z3950/agency/) version 3 support. Amendments and Z39.50-2002 revision is supported.
- Supports SRW (http://www.loc.gov/z3950/agency/zing/srw/) version 1.0 (over HTTP and HTTPS).
- Includes BER encoders/decoders for the ISO ILL (http://www.nlc-bnc.ca/iso/ill/) protocol.
- Supports the following transports: BER over TCP/IP (RFC1729 (http://www.faqs.org/rfcs/rfc1729.html)), BER over unix local socket, and HTTP 1.1 (http://www.w3.org/Protocols/rfc2616/rfc2616.html).
- Secure Socket Layer support using OpenSSL (http://www.openssl.org/). If enabled, YAZ uses HTTPS transport (for SOAP) or "Secure BER" (for Z39.50).
- Offers ZOOM (http://zoom.z3950.org/) C API implementing both Z39.50 and SRW.
- The YAZ library offers a set of useful utilities related to the protocols, such as MARC (ISO2709) parser, CCL (ISO8777) parser, CQL (http://www.loc.gov/z3950/agency/zing/cql/) parser, memory management routines, character set conversion.
- Portable code. YAZ compiles out-of-the box on most Unixes and on Windows using Microsoft Visual C++.
- Fast operation. The C based BER encoders/decoders as well as the server component of YAZ is very fast.
- Liberal license that allows for commercial use of YAZ.

Reading this Manual

Most implementors only need to read a fraction of the material in thie manual, so a quick walkthrough of the chapters is in order.

- Chapter 2 contains installation instructions for YAZ. You don’t need reading this if you expect to download YAZ binaries. However, the chapter contains information about how to make *your* application link with YAZ.
- Chapter 3 describes the ZOOM API of YAZ. This is definitely worth a read if you wish to develop a Z39.50/SRW client.
- Chapter 3 describes the generic frontend server and explains how to develop server Z39.50/SRW applications for YAZ. Obviously worth reading if you’re to develop a server.
- Chapter 5 describes how to use the YAZ Z39.50 client. If you’re developer and wish to test your server or a server from another party, you might find this chapter useful.

- Chapter 6 documents the most commonly used Z39.50 C data structures offered by the YAZ API. Client developers using ZOOM and non-Z39.50 implementors may skip this.
- Chapter 7 describes how SRW and SOAP is used in YAZ. Only if you're developing SOAP/SRW applications this section is a must.
- Chapter 8 contains sections for the various tools offered by YAZ. Scan through the material quickly and see what's relevant to you! SRW implementors might find the CQL section particularly useful.
- Chapter 9 goes through the details of the ODR module which is the work horse that encodes and decodes BER packages. Implementors using ZOOM only do not need reading this. Most other Z39.50 implementors only need to read the first two sections Introduction, Using ODR.
- Chapter 10 describes the network layer module COMSTACK. Implementors using ZOOM or the generic frontend server may skip this. Others, presumably, handling client/server communication on their own should read this.

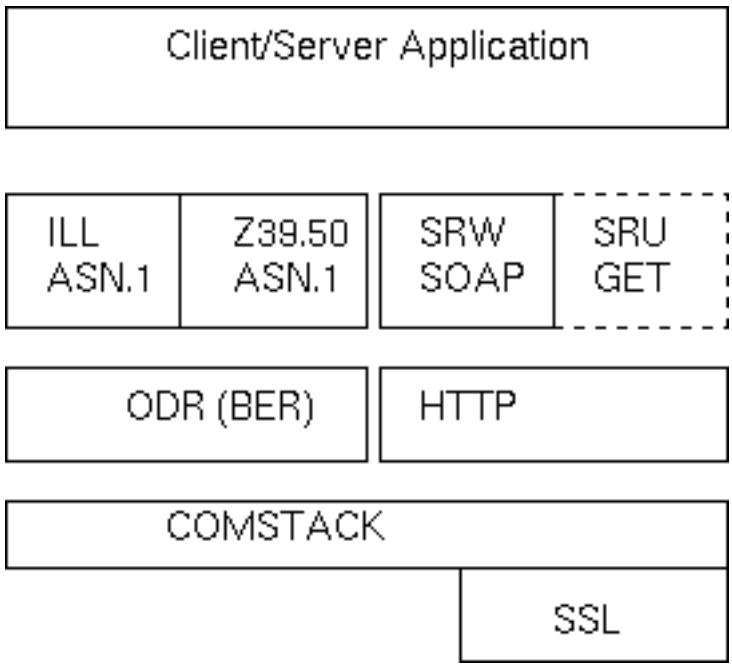
The API

The YAZ (http://www.indexdata.dk/yaz/) toolkit offers several different levels of access to the ISO23950/Z39.50 (http://www.loc.gov/z3950/agency/), ILL (http://www.nlc-bnc.ca/iso/ill/) and SRW (http://www.loc.gov/z3950/agency/zing/srw/) protocols. The level that you need to use depends on your requirements, and the role (server or client) that you want to implement. If you're developing a client application you should consider the ZOOM API. It is, by far, the easiest way to develop clients in C. Server implementers should consider the generic frontend server. None of those high-level APIs support the whole protocol, but they do include most facilities used in existing Z39.50 applications.

If you're using 'exotic' functionality (meaning anything not included in the high-level APIs), developing non-standard extensions to Z39.50 or you're going to develop an ILL application you'll have to learn the lower level APIs of YAZ.

The YAZ toolkit modules is shown in figure Figure 1-1.

Figure 1-1. YAZ layers



There are four layers.

- A client or server application (or both). This layer includes ZOOM and the generic frontend server.
- The second layer provides a C represenation of the protocol units (packages) for Z39.50 ASN.1, ILL ASN.1, SRW SOAP.
- The third layer encodes and decodes protocol data units to simple packages (buffer with certain length). The ODR module encodes and decodes BER whereas the HTTP modules encodes and decodes HTTP ruquests/responses.
- The lowest layer is COMSTACK which exchanges the encoded packages with a peer process over a network.

The Z39.50 ASN.1 module represents the ASN.1 definition of the Z39.50 protocol. It establishes a set of type and structure definitions, with one structure for each of the top-level PDUs, and one structure or type for each of the contained ASN.1 types. For primitive types, or other types that are defined by the ASN.1 standard itself (such as the EXTERNAL type), the C representation is provided by the ODR (Open Data Representation) subsystem.

ODR is a basic mechanism for representing an ASN.1 type in the C programming language, and for implementing BER encoders and decoders for values of that type. The types defined in the Z39.50 ASN.1 module generally have the prefix `Z_`, and a suffix corresponding to the name of the type in the ASN.1 specification of the protocol (generally Z39.50-1995). In the case of base types (those originating in the ASN.1 standard itself), the prefix `Odr_` is sometimes seen. Either way, look for the actual definition in either `z-core.h` (for the types from the protocol), `odr.h` (for the primitive ASN.1 types).

The Z39.50 ASN.1 library also provides functions (which are, in turn, defined using ODR primitives) for encoding and decoding data values. Their general form is

```
int z_xxx(ODR o, Z_xxx **p, int optional, const char *name);
```

(note the lower-case "z" in the function name)

> **Note:** If you are using the premade definitions of the Z39.50 ASN.1 module, and you are not adding new protocol of your own, the only parts of ODR that you need to worry about are documented in section Using ODR.

When you have created a BER-encoded buffer, you can use the COMSTACK subsystem to transmit (or receive) data over the network. The COMSTACK module provides simple functions for establishing a connection (passively or actively, depending on the role of your application), and for exchanging BER-encoded PDUs over that connection. When you create a connection endpoint, you need to specify what transport to use (TCP/IP, SSL or UNIX sockets). For the remainder of the connection's lifetime, you don't have to worry about the underlying transport protocol at all - the COMSTACK will ensure that the correct mechanism is used.

We call the combined interfaces to ODR, Z39.50 ASN.1, and COMSTACK the service level API. It's the API that most closely models the Z39.50 service/protocol definition, and it provides unlimited access to all fields and facilities of the protocol definitions.

The reason that the YAZ service-level API is a conglomerate of the APIs from three different submodules is twofold. First, we wanted to allow the user a choice of different options for each major task. For instance, if you don't like the protocol API provided by ODR/Z39.50 ASN.1, you can use SNACC or BERUtils instead, and still have the benefits of the transparent transport approach of the COMSTACK module. Secondly, we realize that you may have to fit the toolkit into an existing event-processing structure, in a way that is incompatible with the COMSTACK interface or some other part of YAZ.

Chapter 2. Compilation and Installation

Introduction

The latest version of the software will generally be found at:

http://ftp.indexdata.dk/pub/yaz/ (http://ftp.indexdata.dk/pub/yaz/)

We have tried our best to keep the software portable, and on many platforms, you should be able to compile everything with little or no changes. So far, the software has been ported to the following platforms with little or no difficulties.

- Unix systems
  - HP/UX
  - SunOS/Solaris
  - DEC Unix
  - BSDs: FreeBSD, OpenBSD, NetBSD
  - MAC OSX
  - Linux
  - IBM AIX
  - Data General DG/UX (with some CFLAGS tinkering)
  - SGI/IRIX
  - DDE Supermax

- Non-unix systems
  - Apple Macintosh (using the Codewarrior programming environment and the GUSI socket libraries)
  - MS Windows 95/98/NT/2K/XP (Win32)
  - IBM AS/400

If you move the software to other platforms, we'd be grateful if you'd let us know about it. If you run into difficulties, we will try to help if we can, and if you solve the problems, we would be happy to include your fixes in the next release. So far, we have mostly avoided `#ifdefs` for individual platforms, and we'd like to keep it that way as far as it makes sense.

We maintain a mailing-list for the purpose of announcing new releases and bug-fixes, as well as general discussion. Subscribe by sending mail to yaz-request@indexdata.dk (mailto:yaz-request@indexdata.dk) or fill-in the form here (http://www.indexdata.dk/mailman/listinfo/yazlist). General questions and problems can be directed at yaz-help@indexdata.dk (mailto:yaz-help@indexdata.dk), or the address given at the top of this document.

UNIX

We provide Debian GNU/Linux (http://www.debian.org/) and Redhat (http://www.redhat.com/) packages for YAZ. Only i386 binary packages are available. You should be able to create packages for other CPU's by building them from the source package.

Compiling from source on Unix

Note that if your system doesn't have a native ANSI C compiler, you may have to acquire one separately. We recommend GCC (http://gcc.gnu.org/).

If you wish to use character set conversion facilities in YAZ or if you are compiling YAZ for use with Zebra it is a good idea to ensure that the iconv library is installed. Some Unixes today already have it - if not, we suggest GNU iconv (http://www.gnu.org/software/libiconv/).

The XML C library libxml2 (http://www.xmlsoft.org/) is required if YAZ is to support SRW (and SOAP). This library is very portable and should compile out-of-the box on virtually all Unix platforms. It is available in binary forms for Linux and others.

The GNU tools Autoconf (http://www.gnu.org/software/autoconf/), Automake (http://www.gnu.org/software/automake/) and Libtool (http://www.gnu.org/software/libtool/) are used to generate Makefiles and configure YAZ for the system. You do *not* these tools unless you're using the CVS version of YAZ.

The CQL parser for YAZ is built using GNU Bison (http://www.gnu.org/software/bison/). This tool is only needed if you're using the CVS version of YAZ.

YAZ includes a tiny ASN.1 compiler. This compiler is written in Tcl (http://www.tcl.tk/). But as for Bison you do not need it unless you're using CVS version of YAZ or you're using the compiler to built own codecs for private ASN.1.

Generally it should be sufficient to run configure without options, like this:

```
./configure
```

The configure script attempts to use use the C compiler specified by the `CC` environment variable. If not set, GNU C will be used if it is available. The `CFLAGS` environment variable holds options to be passed to the C compiler. If you're using Bourne-compatible shell you may pass something like this to use a particular C compiler with optimization enabled:

```
CC=/opt/ccs/bin/cc CFLAGS=-O ./configure
```

To customize YAZ, the configure script also accepts a set of options. The most important are:

`--prefix=`*`prefix`*

Specifies installation prefix for YAZ. This is only needed if you run `make install` later to perform a "system" installation. The prefix is `/usr/local` if not specified.

`--enable-tcpd`

The front end server will be built using Wietse's TCP wrapper library (ftp://ftp.porcupine.org/pub/security/index.html). It allows you to allow/deny clients depending on IP number. The TCP wrapper library is often used in Linux/BSD distributions. See hosts_access(5) and tcpd(8).

`--enable-threads`

YAZ will be built using POSIX threads. Specifically, `_REENTRANT` will be defined during compilation.

`--enable-shared`

The make process will create shared libraries (also known as shared objects `.so`). By default, no shared libraries are created - equivalent to `--disable-shared`.

`--disable-shared`

The make process will not create static libraries (`.a`). By default, static libraries are created - equivalent to `--enable-static`.

`--with-iconv`[=*`prefix`*]

Compile YAZ with iconv library in directory *`prefix`*. By default configure will search for iconv on your system. Use this option if it doesn't find iconv. Alternatively you can use `--without-iconv` to force YAZ not to use iconv.

`--with-xml2`[=*`prefix`*]

Compile YAZ with libxml2 (http://www.xmlsoft.org/) in directory *`prefix`*. Use this option if you want SOAP support. By default configure will search for libxml2 on your system. Use this option if it doesn't find libxml2. Alternatively you can use `--without-xml2` to force YAZ not to use libxml2.

`--with-openssl`[=*`prefix`*]

YAZ will be linked with the OpenSSL libraries and an SSL COMSTACK will be provided. Note that SSL support is still experimental.

When configured, build the software by typing:

```
make
```

The following files are generated by the make process:

`lib/libyaz.la`

Main YAZ library. This is no ordinary library. It's a Libtool archive. By default, YAZ creates a static library in `lib/.libs/libyaz.a`.

`lib/libyazthread.la`

When threading is supported/enabled by configure this Libtool library is created. It includes functions that allows YAZ to use threads.

`ztest/yaz-ztest`

Test Z39.50 server.

`client/yaz-client`

Z39.50 client for testing the protocol. See chapter YAZ client for more information.

`yaz-config`

A Bourne-shell script, generated by configure, that specifies how external applications should compile - and link with YAZ.

`yaz-comp`

The ASN.1 compiler for YAZ. Requires the Tcl Shell, tclsh, in `PATH` to operate.

`zoom/zoomsh`

A simple shell implemented on top of the ZOOM functions. The shell is a command line application that allows you to enter simple commands to perform ZOOM operations.

`zoom/zoomtst1`, `zoom/zoomtst2`, ..

Several small applications that demonstrates the ZOOM API.

If you wish to install YAZ in system directories `/usr/local/bin`, `/usr/local/lib` .. etc, you can type:

```
make install
```

You probably need to have root access in order to perform this. You must specify the `--prefix` option for configure if you wish to install YAZ in other directories than the default `/usr/local/`.

If you wish to perform an un-installation of YAZ, use:

```
make uninstall
```

This will only work if you haven't reconfigured YAZ (and therefore changed installation prefix). Note that uninstall will not remove directories created by make install, e.g. `/usr/local/include/yaz`.

How to make apps using YAZ on UNIX

This section describes how to compile - and link your own applications using the YAZ toolkit. If you're used to Makefiles this shouldn't be hard. As for other libraries you have used before, you have to set a proper include path for your C/C++ compiler and specify the location of YAZ libraries. You can do it by hand, but generally we suggest you use the `yaz-config` that is generated by `configure`. This is

especially important if you're using the threaded version of YAZ which require you to pass more options to your linker/compiler.

The `yaz-config` script accepts command line options that makes the `yaz-config` script print options that you should use in your make process. The most important ones are: `--cflags`, `--libs` which prints C compiler flags, and linker flags respectively.

A small and complete `Makefile` for a C application consisting of one source file, `myprog.c`, may look like this:

```
YAZCONFIG=/usr/local/bin/yaz-config
CFLAGS=‘$(YAZCONFIG) --cflags‘
LIBS=‘$(YAZCONFIG) --libs‘
myprog: myprog.o
   $(CC) $(CFLAGS) -o myprog myprog.o $(LIBS)
```

The CFLAGS variable consists of a C compiler directive that will set the include path to the *parent* directory of `yaz`. That is, if YAZ header files were installed in `/usr/local/include/yaz`, then include path is set to `/usr/local/include`. Therefore, in your applications you should use

```
#include <yaz/proto.h>
```

and *not*

```
#include <proto.h>
```

For Libtool users, the `yaz-config` script provides a different variant of option `--libs`, called `--lalibs` that returns the name of the Libtool acrhive(s) for YAZ rather than the ordinary ones.

For applications using the threaded version of YAZ, specify `threads` after the other options. When `threads` is given, more flags and linker flags will be printed by `yaz-config`. If our previous example was using threads, you'd have to modify the lines that set `CFLAGS` and `LIBS` as follows:

```
CFLAGS=‘$(YAZCONFIG) --cflags threads‘
LIBS=‘$(YAZCONFIG) --libs threads‘
```

There is no need specify POSIX thread libraries in your Makefile. The `LIBS` variable includes that as well.

WIN32

The easiest way to install YAZ on Windows is by downloading an installer from here (http://ftp.indexdata.dk/pub/yaz/win32). The installer comes with source too - in case you wish to compile YAZ with different Compiler options etc.

Compiling from Source on WIN32

YAZ is shipped with "makefiles" for the NMAKE tool that comes with Microsoft Visual C++ (http://msdn.microsoft.com/vstudio/). Version 6 has been tested. We expect that YAZ compiles with version 5 as well.

Start a command prompt and switch the sub directory `WIN` where the file `makefile` is located. Customize the installation by editing the `makefile` file (for example by using notepad). The following summarizes the most important settings in that file:

`DEBUG`

If set to 1, the software is compiled with debugging libraries (code generation is multi-threaded debug DLL). If set to 0, the software is compiled with release libraries (code generation is multi-threaded DLL).

`HAVE_TCL`, `TCL`

If `HAVE_TCL` is set to 1, nmake will use the ASN.1 compiler (Tcl based). You must set `TCL` to the full path of the Tcl interpreter.

If you do not have Tcl installed, set `HAVE_TCL` to 0.

`HAVE_ICONV`, `ICONV_DIR`

If `HAVE_ICONV` is set to 1, YAZ is compiled with iconv support. In this configuration, set `ICONV_DIR` to the iconv source directory.

`HAVE_LIBXML2`, `LIBXML2_DIR`

If `HAVE_LIBXML2` is set to 1, YAZ is compiled with SRW (and SOAP) support. In this configuration, set `LIBXML2_DIR` to the libxml2 (http://www.xmlsoft.org/) source directory.

You can get libxml2 and iconv binaries from here (http://www.zlatkovic.com/projects/libxml/binaries.html).

When satisfied with the settings in the makefile, type

```
nmake
```

Note: If the `nmake` command is not found on your system you probably haven't defined the environment variables required to use that tool. To fix that, find and run the batch file `vcvars32.bat`. You need to run it from within the command prompt or set the environment variables "globally"; otherwise it doesn't work.

If you wish to recompile YAZ - for example if you modify settings in the `makefile` you can delete object files, etc by running.

```
nmake clean
```

The following files are generated upon successful compilation:

`bin/yaz.dll`

YAZ multi-threaded Dynamic Link Library.

`lib/yaz.lib`

Import library for `yaz.dll`.

`bin/yaz-client.exe`

YAZ Z39.50 client application. It's a WIN32 console application. See chapter YAZ client for more information.

`bin/yaz-ztest.exe`

Z39.50 multi-threaded test/example server. It's a WIN32 console application.

`bin/zoomsh.exe`

Simple console application implemented on top of the ZOOM functions. The application is a command line shell that allows you to enter simple commands to perform ZOOM operations.

`bin/zoomtst1.exe`, `bin/zoomtst2.exe`, ..

Several small applications that demonstrates the ZOOM API.

How to make apps using YAZ on WIN32

This section will go though the process of linking your WIN32 applications with YAZ.

Some people are confused by the fact that we use the nmake tool to build YAZ. They think they have to do that too - in order to make their WIN32 applications work with YAZ. The good news is that you don't have to. You can use the integrated environment of Visual Studio if desired for your own application.

When setting up a project or Makefile you have to set the following:

include path

Set it to the `include` directory of YAZ.

import library `yaz.lib`

You must link with this library. It's located in the sub directory `lib` of YAZ.

dynamic link library `yaz.dll`

This DLL must be in your execution path when you invoke your application. Specifically, you should distribute this DLL with your application.

Chapter 3. ZOOM

ZOOM is an acronym for 'Z39.50 Object-Orientation Model' and is an initiative started by Mike Taylor (Mike is from the UK, which explains the peculiar name of the model). The goal of ZOOM is to provide a common Z39.50 client API not bound to a particular programming language or toolkit.

> **Note:** A recent addition to YAZ is SRW support. You can now make SRW ZOOM connections by specifying another scheme for the hostname for a connection.

The lack of a simple Z39.50 client API for YAZ has become more and more apparent over time. So when the first ZOOM specification became available, an implementation for YAZ was quickly developed. For the first time, it is now as easy (or easier!) to develop clients than servers with YAZ. This chapter describes the ZOOM C binding. Before going further, please reconsider whether C is the right programming language for the job. There are other language bindings available for YAZ, and still more are in active development. See the ZOOM web-site (http://zoom.z3950.org/) for more information.

In order to fully understand this chapter you should read and try the example programs `zoomtst1.c`, `zoomtst2.c`, .. in the `zoom` directory.

The C language misses features found in object oriented languages such as C++, Java, etc. For example, you'll have to manually, destroy all objects you create, even though you may think of them as temporary. Most objects has a `_create` - and a `_destroy` variant. All objects are in fact pointers to internal stuff, but you don't see that because of typedefs. All destroy methods should gracefully ignore a `NULL` pointer.

In each of the sections below you'll find a sub section called protocol behavior, that describes how the API maps to the Z39.50 protocol.

Connections

The Connection object is a session with a target.

```
#include <yaz/zoom.h>

ZOOM_connection ZOOM_connection_new (const char *host, int portnum);

ZOOM_connection ZOOM_connection_create (ZOOM_options options);

void ZOOM_connection_connect(ZOOM_connection c, const char *host,
                             int portnum);
void ZOOM_connection_destroy (ZOOM_connection c);
```

Connection objects are created with either function `ZOOM_connection_new` or `ZOOM_connection_create`. The former creates and automatically attempts to establish a network connection with the target. The latter doesn't establish a connection immediately, thus allowing you to specify options before establishing network connection using the function `ZOOM_connection_connect`. If the port number, `portnum`, is zero, the `host` is consulted for a port specification. If no port is given, 210 is used. A colon denotes the beginning of a port number in the host string. If the host string includes a slash, the following part specifies a database for the connection.

You can prefix the host with a scheme followed by colon. The default scheme is `tcp` (Z39.50 protocol). The scheme `http` selects SRW over HTTP.

Connection objects should be destroyed using the function `ZOOM_connection_destroy`.

```
void ZOOM_connection_option_set (ZOOM_connection c,
                                 const char *key,
                                 const char *val);

const char *ZOOM_connection_option_get (ZOOM_connection c,
                                        const char *key);
```

The `ZOOM_connection_option_set` allows you to set an option given by *`key`* to the value *`value`* for the connection. Function `ZOOM_connection_option_get` returns the value for an option given by *`key`*.

Table 3-1. ZOOM Connection Options

Option	Description	Default
implementationName	Name of Your client	none
user	Authentication user name	none
group	Authentication group name	none
pass	Authentication password	none
host	Target host. This setting is "read-only". It's automatically set internally when connecting to a target.	none
proxy	Proxy host	none
async	If true (1) the connection operates in asynchronous operation which means that all calls are non-blocking except `ZOOM_event`.	0
maximumRecordSize	Maximum size of single record.	1 MB
preferredMessageSize	Maximum size of multiple records.	1 MB
lang	Language for negotiation.	none
charset	Character set for negotiation.	none
targetImplementationId	Implementation ID of target.	none
targetImplementationName	Implementation Name of target.	none
targetImplementationVersion	Implementation Version of target.	none

If either option `lang` or `charset` is set, then Character Set and Language Negotiation (http://lcweb.loc.gov/z3950/agency/defns/charneg-3.html) is in effect.

```
int ZOOM_connection_error (ZOOM_connection c, const char **cp,
                           const char **addinfo);
int ZOOM_connection_error_x (ZOOM_connection c, const char **cp,
                             const char **addinfo, const char **dset);
```

Function ZOOM_connection_error checks for errors for the last operation(s) performed. The function returns zero if no errors occurred; non-zero otherwise indicating the error. Pointers *cp* and *addinfo* holds messages for the error and additional-info if passed as non-NULL. Function ZOOM_connection_error_x is an extended version of ZOOM_connection_error that is capable of returning name of diagnostic set in *dset*.

Z39.50 Protocol behavior

The calls ZOOM_connection_new and ZOOM_connection_connect establishes a TCP/IP connection and sends an Initialize Request to the target if possible. In addition, the calls waits for an Initialize Response from the target and the result is inspected (OK or rejected).

If proxy is set then the client will establish a TCP/IP connection with the peer as specified by the proxy host and the hostname as part of the connect calls will be set as part of the Initialize Request. The proxy server will then "forward" the PDU's transparently to the target behind the proxy.

For the authentication parameters, if option user is set and both options group and pass are unset, then Open style authentication is used (Version 2/3) in which case the username is usually followed by a slash, then by a password. If either group or pass is set then idPass authentication (Version 3 only) is used. If none of the options are set, no authentication parameters are set as part of the Initialize Request (obviously).

When option async is 1, it really means that all network operations are postponed (and queued) until the function ZOOM_event is invoked. When doing so it doesn't make sense to check for errors after ZOOM_connection_new is called since that operation "connecting - and init" is still incomplete and the API cannot tell the outcome (yet).

SRW Protocol behavior

The SRW protocol doesn't feature an Init Request, so the connection phase merely establishes a TCP/IP connection with the SOAP service.

None of the ZOOM connection options affect SRW and they are ignored. However, future versions of YAZ might honor implementationName and put that as part of User-Agent header for HTTP requests. The charset, and lang might also affect HTTP headers in future releases.

Queries

Query objects represents queries.

```
ZOOM_query ZOOM_query_create(void);

void ZOOM_query_destroy(ZOOM_query q);

int ZOOM_query_prefix(ZOOM_query q, const char *str);

int ZOOM_query_cql(ZOOM_query s, const char *str);

int ZOOM_query_sortby(ZOOM_query q, const char *criteria);
```

Create query objects using `ZOOM_query_create` and destroy them by calling `ZOOM_query_destroy`. RPN-queries can be specified in PQF notation by using the function `ZOOM_query_prefix`. The `ZOOM_query_cql` specifies a CQL query to be sent to the server/target. More query types will be added in future versions of YAZ, such as CCL to RPN-mapping, native CCL query, etc. In addition to a search, a sort criteria may be set. Function `ZOOM_query_sortby` specifies a sort criteria using the same string notation for sort as offered by the YAZ client.

Protocol behavior

The query object is just an interface for the member Query in the SearchRequest. The sortby-function is an interface to the sortSequence member of the SortRequest.

Result sets

The result set object is a container for records returned from a target.

```
ZOOM_resultset ZOOM_connection_search(ZOOM_connection,
                                      ZOOM_query q);

ZOOM_resultset ZOOM_connection_search_pqf(ZOOM_connection c,
                                          const char *q);

void ZOOM_resultset_destroy(ZOOM_resultset r);
```

Function `ZOOM_connection_search` creates a result set given a connection and query. Destroy a result set by calling `ZOOM_resultset_destroy`. Simple clients may using PQF only may use function `ZOOM_connection_search_pqf` in which case creating query objects is not necessary.

```
void ZOOM_resultset_option_set (ZOOM_resultset r,
                                const char *key,
                                const char *val);

const char *ZOOM_resultset_option_get (ZOOM_resultset r,
                                       const char *key);

size_t ZOOM_resultset_size (ZOOM_resultset r);
```

Functions `ZOOM_resultset_options_set` and `ZOOM_resultset_get` sets and gets an option for a result set similar to `ZOOM_connection_option_get` and `ZOOM_connection_option_set`.

The number of hits also called result-count is returned by function `ZOOM_resultset_size`.

Option	Description	Default

Table 3-2. ZOOM Result set Options

Option	Description	Default
piggyback	True (1) if piggyback should be used in searches; false (0) if not.	1
start	Offset of first record to be retrieved from target. First record has offset 0 unlike the protocol specifications where first record has position 1.	0
count	Number of records to be retrieved.	0
elementSetName	Element-Set name of records. Most targets should honor element set name `B` and `F` for brief and full respectively.	none
preferredRecordSyntax	Preferred Syntax, such as `USMARC`, `SUTRS`, etc.	none
schema	Schema for retrieval, such as `Gils-schema`, `Geo-schema`, etc.	none
smallSetUpperBound	If hits is less than or equal to this value, then target will return all records using small element set name	0
largeSetLowerBound	If hits is greater than this value, the target will return no records.	1
mediumSetPresentNumber	This value represents the number of records to be returned as part of a search when when hits is less than or equal to large set lower bound and if hits is greater than small set upper bound.	0
smallSetElementSetName	The element set name to be used for small result sets.	none
mediumSetElementSetName	The element set name to be for medium-sized result sets.	none
databaseName	One or more database names separated by character plus (+).	Default
setname	Name of Result Set (Result Set ID). If this option isn't set, the ZOOM module will automatically allocate a result set name.	default

Z39.50 Protocol behavior

The creation of a result set involves at least a SearchRequest - SearchResponse protocol handshake. Following that, if a sort criteria was specified as part of the query, a SortRequest - SortResponse handshake takes place. Note that it is necessary to perform sorting before any retrieval takes place, so no records will be returned from the target as part of the SearchResponse because these would be unsorted. Hence, piggyback is disabled when sort criteria is set. Following Search - and a Possible sort, Retrieval takes place - as one or more Present Requests - Present Response being transferred.

The API allows for two different modes for retrieval. A high level mode which is somewhat more powerful and a low level one. The low level is "enabled" when the settings `smallSetUpperBound`, `mediumSetPresentNumber` and `largeSetLowerBound` are set. The low level mode thus allows you to precisely set how records are returned as part of a search response as offered by the Z39.50 protocol.

Since the client may be retrieving records as part of the search response, this mode doesn't work well if sorting is used.

The high-level mode allows you to fetch a range of records from the result set with a given start offset. When you use this mode the client will automatically use piggyback if that is possible with the target and perform one or more present requests as needed. Even if the target returns fewer records as part of a present response because of a record size limit, etc. the client will repeat sending present requests. As an example, if option `start` is 0 (default) and `count` is 4, and `piggyback` is 1 (default) and no sorting criteria is specified, then the client will attempt to retrieve the 4 records as part the search response (using piggyback). On the other hand, if either `start` is positive or if a sorting criteria is set, or if `piggyback` is 0, then the client will not perform piggyback but send Present Requests instead.

If either of the options `mediumSetElementSetName` and `smallSetElementSetName` are unset, the value of option `elementSetName` is used for piggyback searches. This means that for the high-level mode you only have to specify one elementSetName option rather than three.

SRW Protocol behavior

Current version of YAZ does not take advantage of a result set id returned by the SRW server. Future versions might do, however. Since, the ZOOM driver does not save result set IDs any present (retrieval) is transformed to a SRW SearchRetrieveRequest with same query but, possibly, different offsets.

Option `schema` specifies SRW schema for retrieval. However, options `elementSetName` and `preferredRecordSyntax` are ignored.

Options `start` and `count` are supported by SRW. The remaining options `piggyback`, `smallSetUpperBound`, `largeSetLowerBound`, `mediumSetPresentNumber`, `mediumSetElementSetName`, `smallSetElementSetName` are unsupported.

SRW supports CQL queries, *not* PQF. If PQF is used, however, the PQF query is transferred anyway using non-standard element `pQuery` in SRW SearchRetrieveRequest.

Unfortunately, SRW does not define a database setting. Hence, `databaseName` is unsupported and ignored. However, the path part in host parameter for functions `ZOOM_connecton_new` and `ZOOM_connection_connect` acts as a database (at least for the YAZ SRW server).

Records

A record object is a retrieval record on the client side - created from result sets.

```
void ZOOM_resultset_records (ZOOM_resultset r,
                             ZOOM_record *recs,
                             size_t start, size_t count);
ZOOM_record ZOOM_resultset_record (ZOOM_resultset s, size_t pos);

const char *ZOOM_record_get (ZOOM_record rec, const char *type,
                             size_t *len);

ZOOM_record ZOOM_record_clone (ZOOM_record rec);
```

```
void ZOOM_record_destroy (ZOOM_record rec);
```

References to temporary records are returned by functions `ZOOM_resultset_records` or `ZOOM_resultset_record`.

If a persistent reference to a record is desired `ZOOM_record_clone` should be used. It returns a record reference that should be destroyed by a call to `ZOOM_record_destroy`.

A single record is returned by function `ZOOM_resultset_record` that takes a position as argument. First record has position zero. If no record could be obtained `NULL` is returned.

Function `ZOOM_resultset_records` retrieves a number of records from a result set. Parameter `start` and `count` specifies the range of records to be returned. Upon completion array `recs[0], ..recs[count-1]` holds record objects for the records. The array of records `recs` should be allocated prior the call `ZOOM_resultset_records`. Note that for those records that couldn't be retrieved from the target `recs[ ..]` is set to `NULL`.

In order to extract information about a single record, `ZOOM_record_get` is provided. The function returns a pointer to certain record information. The nature (type) of the pointer depends on the parameter, `type`. In addition, for certain types, the length `len` passed will be set to the size in bytes of the returned information.

`database`

Database of record is returned as a C null-terminated string. Return type `const char *`.

`syntax`

The transfer syntax (OID) of the record is returned as a C null-terminated string. Return type is `const char *`.

`render`

The record is returned in a display friendly format. Upon completion buffer is returned (type `const char *`) and length is stored in `*len`.

`raw`

The record is returned in the internal YAZ specific format. For GRS-1, Explain, and others, the raw data is returned as type `Z_External *` which is just the type for the member `retrievalRecord` in type `NamePlusRecord`. For SUTRS and octet aligned record (including all MARCs) the octet buffer is returned and the length of the buffer.

Z39.50 Protocol behavior

The functions `ZOOM_resultset_record` and `ZOOM_resultset_records` inspects the client-side record cache. Records not found in cache are fetched using Present. The functions may block (and perform network I/O) - even though option `async` is 1, because they return records objects. (and there's no way to return records objects without retrieving them!).

There is a trick, however, in the usage of function `ZOOM_resultset_records` that allows for delayed retrieval (and makes it non-blocking). By passing a null pointer for *`recs`* you're indicating you're not interested in getting records objects *now*.

SRW Protocol behavior

The ZOOM driver for SRW treats records returned by a SRW server as if they where Z39.50 records with transfer syntax XML and no element set name or database name.

Scan

This section describes an interface for Scan. Scan is not an official part of the ZOOM model yet. The result of a scan operation is the `ZOOM_scanset` which is a set of terms returned by a target.

The Scan interface is Z39.50 only. SRW version 1.0 does not support this.

```
ZOOM_scanset ZOOM_connection_scan (ZOOM_connection c,
                                   const char *startterm);

size_t ZOOM_scanset_size(ZOOM_scanset scan);

const char * ZOOM_scanset_term(ZOOM_scanset scan, size_t pos,
                               int *occ, size_t *len);


void ZOOM_scanset_destroy (ZOOM_scanset scan);

const char *ZOOM_scanset_option_get (ZOOM_scanset scan,
                                     const char *key);

void ZOOM_scanset_option_set (ZOOM_scanset scan, const char *key,
                              const char *val);
```

The scan set is created by function `ZOOM_connection_scan` which performs a scan operation on the connection and start term given. If the operation was successful, the size of the scan set can be retrieved by a call to `ZOOM_scanset_size`. Like result sets, the items are numbered 0,..size-1. To obtain information about a particular scan term, call function `ZOOM_scanset_term`. This function takes a scan set offset `pos` and returns a pointer to an actual term or `NULL` if non-present. If present, the `occ` and `len` are set to the number of occurrences and the length of the actual term respectively. A scan set may be freed by a call to function `ZOOM_scanset_destroy`. Functions `ZOOM_scanset_option_get` and `ZOOM_scanset_option_set` retrieves and sets an option respectively.

Table 3-3. ZOOM Scan Set Options

Option	Description	Default
number	Number of Scan Terms requested in next scan. After scan it holds the actual number of terms returned.	10

Option	Description	Default
position	Preferred Position of term in response in next scan; actual position after completion of scan.	1
stepSize	Step Size	0
scanStatus	An integer indicating the Scan Status of last scan.	0

Options

Most ZOOM objects provide a way to specify options to change behavior. From an implementation point of view a set of options is just like an associative array / hash array, etc.

```
ZOOM_options ZOOM_options_create (void);

ZOOM_options ZOOM_options_create_with_parent (ZOOM_options parent);

void ZOOM_options_destroy (ZOOM_options opt);


const char *ZOOM_options_get (ZOOM_options opt, const char *name);

void ZOOM_options_set (ZOOM_options opt, const char *name,
                       const char *v);


typedef const char *(*ZOOM_options_callback)
                                (void *handle, const char *name);

ZOOM_options_callback
        ZOOM_options_set_callback (ZOOM_options opt,
                                   ZOOM_options_callback c,
                                   void *handle);
```

Events

If you're developing non-blocking applications, you have to deal with events.

```
int ZOOM_event (int no, ZOOM_connection *cs);
```

The `ZOOM_event` executes pending events for a number of connections. Supply the number of connections in `no` and an array of connections in `cs` (`cs[0] ... cs[no-1]`). A pending event could be a sending a search, receiving a response, etc. When an event has occurred for one of the connections, this function returns a positive integer `n` denoting that an event occurred for connection `cs[n-1]`. When no events are pending for the connections, a value of zero is returned. To ensure that all outstanding requests are performed call this function repeatedly until zero is returned.

Chapter 4. Generic server

Introduction

If you aren't into documentation, a good way to learn how the back end interface works is to look at the `backend.h` file. Then, look at the small dummy-server in `ztest/ztest.c`. The `backend.h` file also makes a good reference, once you've chewed your way through the prose of this file.

If you have a database system that you would like to make available by means of Z39.50 or SRW, YAZ basically offers your two options. You can use the APIs provided by the Z39.50 ASN.1, ODR, and COMSTACK modules to create and decode PDUs, and exchange them with a client. Using this low-level interface gives you access to all fields and options of the protocol, and you can construct your server as close to your existing database as you like. It is also a fairly involved process, requiring you to set up an event-handling mechanism, protocol state machine, etc. To simplify server implementation, we have implemented a compact and simple, but reasonably full-functioned server-frontend that will handle most of the protocol mechanics, while leaving you to concentrate on your database interface.

Note: The backend interface was designed in anticipation of a specific integration task, while still attempting to achieve some degree of generality. We realize fully that there are points where the interface can be improved significantly. If you have specific functions or parameters that you think could be useful, send us a mail (or better, sign on to the mailing list referred to in the top-level README file). We will try to fit good suggestions into future releases, to the extent that it can be done without requiring too many structural changes in existing applications.

Note: The YAZ server does not provide full SRW functionality. However, it provides an adapter for SRW (to Z39.50).

The Database Frontend

We refer to this software as a generic database frontend. Your database system is the *backend database*, and the interface between the two is called the *backend API*. The backend API consists of a small number of function handlers and structure definitions. You are required to provide the `main()` routine for the server (which can be quite simple), as well as a set of handlers to match each of the prototypes. The interface functions that you write can use any mechanism you like to communicate with your database system: You might link the whole thing together with your database application and access it by function calls; you might use IPC to talk to a database server somewhere; or you might link with third-party software that handles the communication for you (like a commercial database client library). At any rate, the handlers will perform the tasks of:

- Initialization.
- Searching.
- Fetching records.

- Scanning the database index (optional - if you wish to implement SCAN).
- Extended Services (optional).
- Result-Set Delete (optional).
- Result-Set Sort (optional).

(more functions will be added in time to support as much of Z39.50-1995 as possible).

The Backend API

The header file that you need to use the interface are in the `include/yaz` directory. It's called `backend.h`. It will include other files from the `include/yaz` directory, so you'll probably want to use the -I option of your compiler to tell it where to find the files. When you run `make` in the top-level YAZ directory, everything you need to create your server is to link with the `lib/libyaz.la` library.

Your main() Routine

As mentioned, your `main()` routine can be quite brief. If you want to initialize global parameters, or read global configuration tables, this is the place to do it. At the end of the routine, you should call the function

```
int statserv_main(int argc, char **argv,
                  bend_initresult *(*bend_init)(bend_initrequest *r),
                  void (*bend_close)(void *handle));
```

The third and fourth arguments are pointers to handlers. Handler `bend_init` is called whenever the server receives an Initialize Request, so it serves as a Z39.50 session initializer. The `bend_close` handler is called when the session is closed.

`statserv_main` will establish listening sockets according to the parameters given. When connection requests are received, the event handler will typically `fork()` and create a sub-process to handle a new connection. Alternatively the server may be setup to create threads for each connection. If you do use global variables and forking, you should be aware, then, that these cannot be shared between associations, unless you explicitly disable forking by command line parameters.

The server provides a mechanism for controlling some of its behavior without using command-line options. The function

```
statserv_options_block *statserv_getcontrol(void);
```

will return a pointer to a `struct statserv_options_block` describing the current default settings of the server. The structure contains these elements:

`int dynamic`

A boolean value, which determines whether the server will fork on each incoming request (TRUE), or not (FALSE). Default is TRUE. This flag is only read by UNIX-based servers (WIN32 based servers doesn't fork).

`int threads`

A boolean value, which determines whether the server will create a thread on each incoming request (TRUE), or not (FALSE). Default is FALSE. This flag is only read by UNIX-based servers that offer POSIX Threads support. WIN32-based servers always operate in threaded mode.

`int inetd`

A boolean value, which determines whether the server will operates under a UNIX INET daemon (inetd). Default is FALSE.

`int loglevel`

Set this by ORing the constants defined in `include/yaz/yaz-log.h`.

`char logfile[ODR_MAXNAME+1]`

File for diagnostic output ("": stderr).

`char apdufile[ODR_MAXNAME+1]`

Name of file for logging incoming and outgoing APDUs ("": don't log APDUs, "-": `stderr`).

`char default_listen[1024]`

Same form as the command-line specification of listener address. "": no default listener address. Default is to listen at "tcp:@:9999". You can only specify one default listener address in this fashion.

`enum oid_proto default_proto;`

Either `PROTO_Z3950` or `PROTO_SR`. Default is `PROTO_Z39_50`.

`int idle_timeout;`

Maximum session idle-time, in minutes. Zero indicates no (infinite) timeout. Default is 120 minutes.

`int maxrecordsize;`

Maximum permissible record (message) size. Default is 1Mb. This amount of memory will only be allocated if a client requests a very large amount of records in one operation (or a big record). Set it to a lower number if you are worried about resource consumption on your host system.

`char configname[ODR_MAXNAME+1]`

Passed to the backend when a new connection is received.

`char setuid[ODR_MAXNAME+1]`

Set user id to the user specified, after binding the listener addresses.

`void (*bend_start)(struct statserv_options_block *p)`

Pointer to function which is called after the command line options have been parsed - but before the server starts listening. For forked UNIX servers this handler is called in the mother process; for threaded servers this handler is called in the main thread. The default value of this pointer is NULL in which case it isn't invoked by the frontend server. When the server operates as an NT service this handler is called whenever the service is started.

`void (*bend_stop)(struct statserv_options_block *p)`

Pointer to function which is called whenever the server has stopped listening for incoming connections. This function pointer has a default value of NULL in which case it isn't called. When the server operates as an NT service this handler is called whenever the service is stopped.

`void *handle`

User defined pointer (default value NULL). This is a per-server handle that can be used to specify "user-data". Do not confuse this with the session-handle as returned by bend_init.

The pointer returned by `statserv_getcontrol` points to a static area. You are allowed to change the contents of the structure, but the changes will not take effect before you call

```
void statserv_setcontrol(statserv_options_block *block);
```

Note: that you should generally update this structure before calling `statserv_main()`.

The Backend Functions

For each service of the protocol, the backend interface declares one or two functions. You are required to provide implementations of the functions representing the services that you wish to implement.

Init

```
bend_initresult (*bend_init)(bend_initrequest *r);
```

This handler is called once for each new connection request, after a new process/thread has been created, and an Initialize Request has been received from the client. The pointer to the `bend_init` handler is passed in the call to `statserv_start`.

This handler is also called when operating in SRW mode - when a connection has been made (even though SRW does not offer this service).

Unlike previous versions of YAZ, the `bend_init` also serves as a handler that defines the Z39.50 services that the backend wish to support. Pointers to *all* service handlers, including search - and fetch must be specified here in this handler.

The request - and result structures are defined as

```
typedef struct bend_initrequest
{
    Z_IdAuthentication *auth;
    ODR stream;               /* encoding stream */
    ODR print;                /* printing stream */
    Z_ReferenceId *referenceId;/* reference ID */
    char *peer_name;          /* dns host of peer (client) */

    char *implementation_id;
    char *implementation_name;
    char *implementation_version;
    int (*bend_sort) (void *handle, bend_sort_rr *rr);
    int (*bend_search) (void *handle, bend_search_rr *rr);
    int (*bend_fetch) (void *handle, bend_fetch_rr *rr);
    int (*bend_present) (void *handle, bend_present_rr *rr);
    int (*bend_esrequest) (void *handle, bend_esrequest_rr *rr);
    int (*bend_delete)(void *handle, bend_delete_rr *rr);
    int (*bend_scan)(void *handle, bend_scan_rr *rr);
    int (*bend_segment)(void *handle, bend_segment_rr *rr);

    ODR decode;                /* decoding stream */
    /* character set and language negotiation - see include/yaz/z-charneg.h */
    Z_CharSetandLanguageNegotiation *charneg_request;
    Z_External *charneg_response;
} bend_initrequest;

typedef struct bend_initresult
{
    int errcode;       /* 0==OK */
    char *errstring;   /* system error string or NULL */
    void *handle;      /* private handle to the backend module */
} bend_initresult;
```

In general, the server frontend expects that the `bend_*result` pointer that you return is valid at least until the next call to a `bend_* function`. This applies to all of the functions described herein. The parameter structure passed to you in the call belongs to the server frontend, and you should not make assumptions about its contents after the current function call has completed. In other words, if you want to retain any of the contents of a request structure, you should copy them.

The `errcode` should be zero if the initialization of the backend went well. Any other value will be interpreted as an error. The `errstring` isn't used in the current version, but one option would be to stick it in the initResponse as a VisibleString. The `handle` is the most important parameter. It should be set to some value that uniquely identifies the current session to the backend implementation. It is used by the frontend server in any future calls to a backend function. The typical use is to set it to point to a dynamically allocated state structure that is private to your backend module.

The `auth` member holds the authentication information part of the Z39.50 Initialize Request. Interpret this if your serves requires authentication.

The members `peer_name`, `implementation_id`, `implementation_name` and `implementation_version` holds DNS of client, ID of implementor, name of client (Z39.50) implementation - and version.

The `bend_` - members are set to NULL when `bend_init` is called. Modify the pointers by setting them to point to backend functions.

Search and retrieve

We now describe the handlers that are required to support search - and retrieve. You must support two functions - one for search - and one for fetch (retrieval of one record). If desirable you can provide a third handler which is called when a present request is received which allows you to optimize retrieval of multiple-records.

```
int (*bend_search) (void *handle, bend_search_rr *rr);

typedef struct {
    char *setname;             /* name to give to this set */
    int replace_set;           /* replace set, if it already exists */
    int num_bases;             /* number of databases in list */
    char **basenames;          /* databases to search */
    Z_ReferenceId *referenceId;/* reference ID */
    Z_Query *query;            /* query structure */
    ODR stream;                /* encode stream */
    ODR decode;                /* decode stream */
    ODR print;                 /* print stream */

    bend_request request;
    bend_association association;
    int *fd;
    int hits;                  /* number of hits */
    int errcode;               /* 0==OK */
    char *errstring;           /* system error string or NULL */
} bend_search_rr;
```

The `bend_search` handler is a fairly close approximation of a protocol Z39.50 Search Request - and Response PDUs The `setname` is the resultSetName from the protocol. You are required to establish a mapping between the set name and whatever your backend database likes to use. Similarly, the `replace_set` is a boolean value corresponding to the resultSetIndicator field in the protocol. `num_bases/basenames` is a length of/array of character pointers to the database names provided by the client. The `query` is the full query structure as defined in the protocol ASN.1 specification. It can be either of the possible query types, and it's up to you to determine if you can handle the provided query type. Rather than reproduce the C interface here, we'll refer you to the structure definitions in the file `include/yaz/z-core.h`. If you want to look at the attributeSetId OID of the RPN query, you can either match it against your own internal tables, or you can use the `oid_getentbyoid` function provided by YAZ.

The structure contains a number of hits, and an `errcode/errstring` pair. If an error occurs during the search, or if you're unhappy with the request, you should set the errcode to a value from the BIB-1 diagnostic set. The value will then be returned to the user in a nonsurrogate diagnostic record in the

response. The `errstring`, if provided, will go in the addinfo field. Look at the protocol definition for the defined error codes, and the suggested uses of the addinfo field.

The `bend_search` handler is also called when the frontend server receives a SRW SearchRetrieveRequest. For SRW, a CQL query is usually provided by the client. The CQL query is available as part of `Z_Query` structure (note that CQL is now part of Z39.50 via an external). To support CQL in existing implementations that only do Type-1, we refer to the CQL-to-PQF tool described here.

To maintain backwards compatibility, the frontend server of yaz always assume that error codes are BIB-1 diagnostics. For SRW operation, a Bib-1 diagnostic code is mapped to SRW diagnostic.

```
int (*bend_fetch) (void *handle, bend_fetch_rr *rr);

typedef struct bend_fetch_rr {
    char *setname;             /* set name */
    int number;                /* record number */
    Z_ReferenceId *referenceId;/* reference ID */
    oid_value request_format;  /* One of the CLASS_RECSYN members */
    int *request_format_raw;   /* same as above (raw OID) */
    Z_RecordComposition *comp; /* Formatting instructions */
    ODR stream;                /* encoding stream - memory source if req */
    ODR print;                 /* printing stream */

    char *basename;            /* name of database that provided record */
    int len;                   /* length of record or -1 if structured */
    char *record;              /* record */
    int last_in_set;           /* is it?  */
    oid_value output_format;   /* format */
    int *output_format_raw;    /* used instead of above if not-null */
    int errcode;               /* 0==success */
    char *errstring;           /* system error string or NULL */
    int surrogate_flag;        /* surrogate diagnostic */
} bend_fetch_rr;
```

The frontend server calls the `bend_fetch` handler when it needs database records to fulfill a Z39.50 Search Request, a Z39.50 Present Request or a SRW SearchRetrieveRequest. The `setname` is simply the name of the result set that holds the reference to the desired record. The `number` is the offset into the set (with 1 being the first record in the set). The `format` field is the record format requested by the client (See section Object Identifiers). The value `VAL_NONE` indicates that the client did not request a specific format. The `stream` argument is an ODR stream which should be used for allocating space for structured data records. The stream will be reset when all records have been assembled, and the response package has been transmitted. For unstructured data, the backend is responsible for maintaining a static or dynamic buffer for the record between calls.

If a SRW SearchRetrieveRequest is received by the frontend server, the `referenceId` is NULL, the `request_format` (transfer syntax) is XML (OID name `VAL_TEXT_XML`). The schema for SRW is stored in the `Z_RecordComposition` structure.

In the structure, the `basename` is the name of the database that holds the record. `len` is the length of the record returned, in bytes, and `record` is a pointer to the record. `Last_in_set` should be nonzero only if the record returned is the last one in the given result set. `errcode` and `errstring`, if given, will be interpreted as a global error pertaining to the set, and will be returned in a non-surrogate-diagnostic. If

you wish to return the error as a surrogate-diagnostic (local error) you can do this by setting `surrogate_flag` to 1 also.

If the `len` field has the value -1, then `record` is assumed to point to a constructed data type. The `format` field will be used to determine which encoder should be used to serialize the data.

> **Note:** If your backend generates structured records, it should use `odr_malloc()` on the provided stream for allocating data: This allows the frontend server to keep track of the record sizes.

The `format` field is mapped to an object identifier in the direct reference of the resulting EXTERNAL representation of the record.

> **Note:** The current version of YAZ only supports the direct reference mode.

```
int (*bend_present) (void *handle, bend_present_rr *rr);

typedef struct {
    char *setname;             /* set name */
    int start;
    int number;                /* record number */
    oid_value format;          /* One of the CLASS_RECSYN members */
    Z_ReferenceId *referenceId;/* reference ID */
    Z_RecordComposition *comp; /* Formatting instructions */
    ODR stream;                /* encoding stream */
    ODR print;                 /* printing stream */
    bend_request request;
    bend_association association;

    int hits;                  /* number of hits */
    int errcode;               /* 0==OK */
    char *errstring;           /* system error string or NULL */
} bend_present_rr;
```

The `bend_present` handler is called when the server receives a Z39.50 Present Request. The `setname`, `start` and `number` is the name of the result set - start position - and number of records to be retrieved respectively. `format` and `comp` is the preferred transfer syntax and element specifications of the present request.

Note that this is handler serves as a supplement for `bend_fetch` and need not to be defined in order to support search - and retrieve.

Delete

For back-ends that supports delete of a result set only one handler must be defined.

```
int (*bend_delete)(void *handle, bend_delete_rr *rr);

typedef struct bend_delete_rr {
```

```
    int function;
    int num_setnames;
    char **setnames;
    Z_ReferenceId *referenceId;
    int delete_status;      /* status for the whole operation */
    int *statuses;          /* status each set - indexed as setnames */
    ODR stream;
    ODR print;
} bend_delete_rr;
```

Note: The delete set function definition is rather primitive, mostly because we have had no practical need for it as of yet. If someone wants to provide a full delete service, we'd be happy to add the extra parameters that are required. Are there clients out there that will actually delete sets they no longer need?

scan

For servers that wish to offer the scan service one handler must be defined.

```
int (*bend_delete)(void *handle, bend_delete_rr *rr);

typedef enum {
    BEND_SCAN_SUCCESS,  /* ok */
    BEND_SCAN_PARTIAL   /* not all entries could be found */
} bend_scan_status;

typedef struct bend_scan_rr {
    int num_bases;      /* number of elements in database list */
    char **basenames;   /* databases to search */
    oid_value attributeset;
    Z_ReferenceId *referenceId; /* reference ID */
    Z_AttributesPlusTerm *term;
    ODR stream;         /* encoding stream - memory source if required */
    ODR print;          /* printing stream */

    int *step_size;     /* step size */
    int term_position;  /* desired index of term in result list/returned */
    int num_entries;    /* number of entries requested/returned */

    struct scan_entry *entries;
    bend_scan_status status;
    int errcode;
    char *errstring;
} bend_scan_rr;
```

Application Invocation

The finished application has the following invocation syntax (by way of `statserv_main()`):

appname [`-a` *`file`*][`-v` *`level`*][`-l` *`file`*][`-u` *`uid`*][`-c` *`config`*][`-t` *`minutes`*] [`-k` *`kilobytes`*][`-d` *`daemon`*][`-w` *`dir`*][`-ziST1`] [listener-spec...]

The options are:

`-a` *`file`*

Specify a file for dumping PDUs (for diagnostic purposes). The special name `-` (dash) sends output to `stderr`.

`-S`

Don’t fork or make threads on connection requests. This is good for debugging, but not recommended for real operation: Although the server is asynchronous and non-blocking, it can be nice to keep a software malfunction (okay then, a crash) from affecting all current users.

`-1`

Like `-S` but after one session the server exits. This mode is for debugging *only*.

`-T`

Operate the server in threaded mode. The server creates a thread for each connection rather than a fork a process. Only available on UNIX systems that offers POSIX threads.

`-s`

Use the SR protocol (obsolete).

`-z`

Use the Z39.50 protocol (default). This option and `-s` complement each other. You can use both multiple times on the same command line, between listener-specifications (see below). This way, you can set up the server to listen for connections in both protocols concurrently, on different local ports.

`-l` *`file`*

The logfile.

`-c` *`config`*

A user option that serves as a specifier for some sort of configuration, usually a filename. The argument to this option is transferred to member `confignameof` the `statserv_options_block`.

`-v` *`level`*

The log level. Use a comma-separated list of members of the set {fatal,debug,warn,log,malloc,all,none}.

`-u` *`uid`*

Set user ID. Sets the real UID of the server process to that of the given user. It's useful if you aren't comfortable with having the server run as root, but you need to start it as such to bind a privileged port.

`-w` *`dir`*

The server changes to this directory during before listening on incoming connections. This option is useful when the server is operating from the inetd daemon (see `-i`).

`-i`

Use this to make the the server run from the inetd server (UNIX only).

`-install`

Use this to install the server as an NT service (Windows 2000/NT only). Control the server by going to the Services in the Control Panel.

`-remove`

Use this to remove the server from the NT services (Windows 2000/NT only).

`-t` *`minutes`*

Idle session timeout, in minutes.

`-k` *`size`*

Maximum record size/message size, in kilobytes.

`-d` *`daemon`*

Set name of daemon to be used in hosts access file. See hosts_access(5) and tcpd(8).

A listener specification consists of a transport mode followed by a colon (:) followed by a listener address. The transport mode is either `tcp`, `unix:` or `ssl`.

For TCP and SSL, an address has the form

```
hostname | IP-number [: portnumber]
```

The port number defaults to 210 (standard Z39.50 port).

For UNIX, the address is the filename of socket.

For TCP/IP and SSL, the special hostname "@" is mapped to the address `INADDR_ANY`, which causes the server to listen on any local interface.

Examples:

```
tcp:@:210
```

```
ssl:@:3000
```

```
unix:/tmp/yaz
```

Chapter 5. The YAZ client

Introduction

yaz-client is a line-mode Z39.50 client. It supports a fair amount of the functionality of the Z39.50-1995 standard Its primary purpose is to exercise the package, and verify that the protocol works OK. For the same reason some commands offers more functionality than others. Commands that exercises common Z39.50 services such as search and present have more features than less common supported services, such as Extended Services (ItemOrder, ItemUpdate,..).

Invoking the YAZ client

It can be started by typing

yaz-client [-m *fname*] [-a *fname*] [-c *fname*] [-v *level*] [-p *target*] [-u *auth*] [-k *size*] [zurl]

in a UNIX shell / WIN32 console. The *zurl*, specifies a Z39.50 host and, if specified, the client first tries to establish connection with the Z39.50 target on the host. Options are prefixed by `-` followed by a particular letter.

The following options are supported:

`-m` *fname*

All retrieved transfer records are appended to file *fname*. All records as returned by a target(s) in Search Responses and Present Responses are appended verbatim to the file.

`-a` *fname*

Pretty-print log of APDUs sent and received is appended to the file *fname*. If *fname* is `-` (minus) the APDU log is written to `stderr`.

`-c` *fname*

Sets the filename for CCL fields to *fname*. If this option is not given the YAZ client reads CCL fields from file `default.bib`.

`-v` *level*

Sets the LOG level to *level*. Level is a sequence of tokens separated by comma. Each token is a integer or a named LOG item - one of `fatal`, `debug`, `warn`, `log`, `malloc`, `all`, `none`.

`-p` *target*

Specifies proxy address. When set YAZ client will connect to a proxy on the address and port given. The actual target will be specified as part of the InitRequest to inform the proxy about actual target.

`-u` *auth*

Specifies authentication. Usually the form *user*/*password* is used. This option does the same thing as the `auth` command.

`-k` *`size`*

Specifies the maximum messages size in kilobytes. The default maximum message size for the YAZ client is 1024 (1 MB).

In order to connect to Index Data's test Z39.50 server on `bagel.indexdata.dk`, port 210 and with the database name marc, one would have to type

```
yaz-client bagel.indexdata.dk:210/marc
```

In order to enable APDU log and connect to localhost, port 210 (default) and database Default (default) you'd write:

```
yaz-client -a - localhost
```

The following command connects to a local server via UNIX socket `/tmp/yaz` and sets maximum message size to 5 MB.

```
yaz-client -k 5120 unix:/tmp/yaz
```

Commands

When the YAZ client has read options and connected to a target, if given, it will display `Z>` and await your command. Commands are executed by hitting the return key. You can always issue the command `?` to see the list of available commands.

The commands are (the letters in parenthesis are short names for the commands):

`open` *`zurl`*

Opens a connection to a server. The syntax for *`zurl`* is the same as described above for connecting from the command line.

Syntax:

`[(tcp|ssl|unix)':']`*`host`* [:*`port`*][/*`base`*>]

`quit`

Ends YAZ client

`f` *`query`*

Sends a Search Request using the *`query`* given.

`delete` *`setname`*

Deletes result set with name *`setname`* on the server.

`base` *`base1 base2 ...`*

Sets the name(s) of the database(s) to search. One or more databases may be specified separated by blanks. This commands overrides the database given in *`zurl`*.

`show` [*`start`*[+*`number`*]]

Fetches records by sending a Present Request from the start position given by *`start`* a number of records given by *`number`*. If *`start`* is not given, then the client will fetch from position of the last retrieved record plus 1. If *`number`* is not given, then one record will be fetched at a time.

`scan` *`term`*

Scans database index for a term. The syntax resembles the syntax for `find`. If you want to scan for the word `water` you could write

```
scan water
```

but if you want to scan only in, say the title field, you would write

```
scan @attr 1=4 water
```

`sort` *`sortspecs`*

Sorts a result set. The sort command takes a sequence of sort specifications. A sort specification holds a field (sort criteria) and is followed by flags. If the sort criteria includes = it is assumed that the sort SortKey is of type sortAttributes using Bib-1. The integer before = is the attribute type and the integer following = is the attribute value. If no = is in the SortKey it is treated as a sortfield-type of type InternationalString. Flags observed are: `s` for case sensitive, `i` for case insensitive, < for sort ascending and > for sort descending.

`sort+`

Same as `sort` but stores the sorted result set in a new result set.

`authentication` *`openauth`*

Sets up a authentication string if a server requires authentication (v2 OpenStyle). The authentication string is first sent to the server when the `open` command is issued and the Z39.50 Initialize Request is sent, so this command must be used before `open` in order to be effective. A common convention for the *`authopen`* string is that the username - and password is separated by a slash, e.g. `myusername/mysecret`.

`lslb` *`n`*

Sets the limit for when no records should be returned together with the search result. See the Z39.50 standard (http://lcweb.loc.gov/z3950/agency/markup/04.html#3.2.2.1.6) for more details.

`ssub` *`n`*

Sets the limit for when all records should be returned with the search result. See the Z39.50 standard (http://lcweb.loc.gov/z3950/agency/markup/04.html#3.2.2.1.6) for more details.

`mspn` *n*

Sets the number of records should be returned if the number of records in the result set is between the values of `lslb` and `ssub`. See the Z39.50 standard (http://lcweb.loc.gov/z3950/agency/markup/04.html#3.2.2.1.6) for more details.

`status`

Displays the values of `lslb`, `ssub` and `mspn`.

`setname`

Switches named result sets on and off. Default is on.

`cancel`

Sends a Trigger Resource Control Request to the target.

`format` *oid*

Sets the preferred transfer syntax for retrieved records. yaz-client supports all the record syntaxes that currently are registered. See Z39.50 Standard (http://lcweb.loc.gov/z3950/agency/defns/oids.html#5) for more details. Commonly used records syntaxes include usmarc, sutrs, grs1 and xml.

`elements` *e*

Sets the element set name for the records. Many targets support element sets are B (for brief) and F (for full).

`close`

Sends a Z39.50 Close APDU and closes connection with the peer

`querytype` *type*

Sets the query type as used by command `find`. The following is supported: `prefix` for Prefix Query Notation (Type-1 Query); `ccl` for CCL search (Type-2 Query) or `ccl2rpn` for CCL to RPN conversion (Type-1 Query).

`attributeset` *set*

Sets attribute set OID for prefix queries (RPN, Type-1).

`refid` *id*

Sets reference ID for Z39.50 Request(s).

`itemorder` *type no*

Sends an Item Order Request using the ILL External. *type* is either 1 or 2 which corresponds to ILL-Profile 1 and 2 respectively. The *no* is the Result Set position of the record to be ordered.

`update`

Sends Item Update Request. This command sends a "minimal" PDU Update to the target supplying the last received record from the target. If no record has been received from the target this command is ignored and nothing is sent to the target.

`. filename`

Executes list of commands from file *`filename`*, just like source on most UNIX shells.

`! args`

Executes command *`args`* in subshell using the `system` call.

`push_commande command`

The push_command takes another command as its argument. That command is then added to the history information (so you can retrieve it later). The command itself is not executed. This command only works if you have GNU readline/history enabled.

`set_apdufile filename`

Sets that APDU should be logged to file *`filename`*. This command does the thing as option `-a`.

`set_marcdump filename`

Specifies that all retrieved records should be appended ot file *`filename`*. This command does the thing as option `-m`.

`set_cclfields filename`

Specifies that CCL fields should be read from file file *`filename`*. This command does the thing as option `-c`.

`register_oid name class OID`

This command allows you to register your own object identifier - so that instead of entering a long dot-notation you can use a short name instead. The *`name`* is your name for the OID, *`class`* is the class, and *`OID`* is the raw OID in dot notation. Class is one `appctx`, `absyn`, `attet`, `transyn`, `diagset`, `recsyn`, `resform`, `accform`, `extserv`, `userinfo`, `elemspec`, `varset`, `schema`, `tagset`, `general`. If you're in doubt use the `general` class.

Searching

The simplest example of a Prefix Query would be something like

```
f knuth
```

or

```
f "donald knuth"
```

In those queries no attributes was specified. This leaves it up to the server what fields to search but most servers will search in all fields. Some servers does not support this feature though, and require that some attributes are defined. To add one attribute you could do:

```
f @attr 1=4 computer
```

where we search in the title field, since the use(1) is title(4). If we want to search in the author field *and* in the title field, and in the title field using right truncation it could look something like this:

```
f @and @attr 1=1003 knuth @attr 1=4 @attr 5=1 computer
```

Finally using a mix of Bib-1 and GILS attributes could look something like this:

```
f @attrset Bib-1 @and @attr GILS 1=2008 Washington @attr 1=21 weather
```

For the full specification of the Prefix Query see the section Prefix Query Format.

Chapter 6. The Z39.50 ASN.1 Module

Introduction

The Z39.50 ASN.1 module provides you with a set of C struct definitions for the various PDUs of the Z39.50 protocol, as well as for the complex types appearing within the PDUs. For the primitive data types, the C representation often takes the form of an ordinary C language type, such as `int`. For ASN.1 constructs that have no direct representation in C, such as general octet strings and bit strings, the ODR module (see section The ODR Module) provides auxiliary definitions.

The Z39.50 ASN.1 module is located in sub directory `z39.50`. There you'll find C files that implements encoders and decoders for the Z39.50 types. You'll also find the protocol definitions: `z3950v3.asn`, `esupdate.asn`, and others.

Preparing PDUs

A structure representing a complex ASN.1 type doesn't in itself contain the members of that type. Instead, the structure contains *pointers* to the members of the type. This is necessary, in part, to allow a mechanism for specifying which of the optional structure (SEQUENCE) members are present, and which are not. It follows that you will need to somehow provide space for the individual members of the structure, and set the pointers to refer to the members.

The conversion routines don't care how you allocate and maintain your C structures - they just follow the pointers that you provide. Depending on the complexity of your application, and your personal taste, there are at least three different approaches that you may take when you allocate the structures.

You can use static or automatic local variables in the function that prepares the PDU. This is a simple approach, and it provides the most efficient form of memory management. While it works well for flat PDUs like the InitReqest, it will generally not be sufficient for say, the generation of an arbitrarily complex RPN query structure.

You can individually create the structure and its members using the `malloc(2)` function. If you want to ensure that the data is freed when it is no longer needed, you will have to define a function that individually releases each member of a structure before freeing the structure itself.

You can use the `odr_malloc()` function (see section Using ODR for details). When you use `odr_malloc()`, you can release all of the allocated data in a single operation, independent of any pointers and relations between the data. `odr_malloc()` is based on a "nibble-memory" scheme, in which large portions of memory are allocated, and then gradually handed out with each call to `odr_malloc()`. The next time you call `odr_reset()`, all of the memory allocated since the last call is recycled for future use (actually, it is placed on a free-list).

You can combine all of the methods described here. This will often be the most practical approach. For instance, you might use `odr_malloc()` to allocate an entire structure and some of its elements, while you leave other elements pointing to global or per-session default variables.

The Z39.50 ASN.1 module provides an important aid in creating new PDUs. For each of the PDU types (say, `Z_InitRequest`), a function is provided that allocates and initializes an instance of that PDU type for you. In the case of the InitRequest, the function is simply named `zget_InitRequest()`, and it sets up reasonable default value for all of the mandatory members. The optional members are generally

initialized to null pointers. This last aspect is very important: it ensures that if the PDU definitions are extended after you finish your implementation (to accommodate new versions of the protocol, say), you won't get into trouble with uninitialized pointers in your structures. The functions use `odr_malloc()` to allocate the PDUs and its members, so you can free everything again with a single call to `odr_reset()`. We strongly recommend that you use the `zget_*` functions whenever you are preparing a PDU (in a C++ API, the `zget_` functions would probably be promoted to constructors for the individual types).

The prototype for the individual PDU types generally look like this:

```
Z_<type> *zget_<type>(ODR o);
```

eg.:

```
Z_InitRequest *zget_InitRequest(ODR o);
```

The ODR handle should generally be your encoding stream, but it needn't be.

As well as the individual PDU functions, a function `zget_APDU()` is provided, which allocates a top-level Z-APDU of the type requested:

```
Z_APDU *zget_APDU(ODR o, int which);
```

The `which` parameter is (of course) the discriminator belonging to the `Z_APDU CHOICE` type. All of the interface described here is provided by the Z39.50 ASN.1 module, and you access it through the `proto.h` header file.

Object Identifiers

When you refer to object identifiers in your application, you need to be aware that SR and Z39.50 use two different set of OIDs to refer to the same objects. To handle this easily, YAZ provides a utility module to Z39.50 ASN.1 which provides an internal representation of the OIDs used in both protocols. Each oid is described by a structure:

```
typedef struct oident
{
    enum oid_proto proto;
    enum oid_class class;
    enum oid_value value;
    int oidsuffix[OID_SIZE];
    char *desc;
} oident;
```

The `proto` field can be set to either `PROTO_SR` or `PROTO_Z3950`. The `class` might be, say, `CLASS_RECSYN`, and the `value` might be `VAL_USMARC` for the USMARC record format. Functions

```
int *oid_ent_to_oid(struct oident *ent, int *dst);
struct oident *oid_getentbyoid(int *o);
```

are provided to map between object identifiers and database entries. If you store a member of the `oid_proto` type in your association state information, it's a simple matter, at runtime, to generate the correct OID when you need it. For decoding, you can simply ignore the proto field, or if you're strict, you can verify that your peer is using the OID family from the correct protocol. The `desc` field is a short, human-readable name for the PDU, useful mainly for diagnostic output.

Note: The old function `oid_getoidbyent` still exists but is not thread safe. Use `oid_ent_to_oid` instead and pass an array of size `OID_SIZE`.

Note: Plans are underway to merge the two protocols into a single definition, with one set of object identifiers. When this happens, the oid module will no longer be required to support protocol independence, but it should still be useful as a simple OID database.

EXTERNAL Data

In order to achieve extensibility and adaptability to different application domains, the new version of the protocol defines many structures outside of the main ASN.1 specification, referencing them through ASN.1 EXTERNAL constructs. To simplify the construction and access to the externally referenced data, the Z39.50 ASN.1 module defines a specialized version of the EXTERNAL construct, called `Z_External`.It is defined thus:

```
typedef struct Z_External
{
    Odr_oid *direct_reference;
    int *indirect_reference;
    char *descriptor;
    enum
    {
        /* Generic types */
        Z_External_single = 0,
        Z_External_octet,
        Z_External_arbitrary,

        /* Specific types */
        Z_External_SUTRS,
        Z_External_explainRecord,
        Z_External_resourceReport1,
        Z_External_resourceReport2

    ...

    } which;
    union
    {
        /* Generic types */
        Odr_any *single_ASN1_type;
```

```
        Odr_oct *octet_aligned;
        Odr_bitmask *arbitrary;

        /* Specific types */
        Z_SUTRS *sutrs;
        Z_ExplainRecord *explainRecord;
        Z_ResourceReport1 *resourceReport1;
        Z_ResourceReport2 *resourceReport2;

        ...

    } u;
} Z_External;
```

When decoding, the Z39.50 ASN.1 module will attempt to determine which syntax describes the data by looking at the reference fields (currently only the direct-reference). For ASN.1 structured data, you need only consult the `which` field to determine the type of data. You can the access the data directly through the union. When constructing data for encoding, you set the union pointer to point to the data, and set the `which` field accordingly. Remember also to set the direct (or indirect) reference to the correct OID for the data type. For non-ASN.1 data such as MARC records, use the `octet_aligned` arm of the union.

Some servers return ASN.1 structured data values (eg. database records) as BER-encoded records placed in the `octet-aligned` branch of the EXTERNAL CHOICE. The ASN-module will *not* automatically decode these records. To help you decode the records in the application, the function

```
Z_ext_typeent *z_ext_gettypebyref(oid_value ref);
```

Can be used to retrieve information about the known, external data types. The function return a pointer to a static area, or NULL, if no match for the given direct reference is found. The `Z_ext_typeent` is defined as:

```
typedef struct Z_ext_typeent
{
    oid_value dref;    /* the direct-reference OID value. */
    int what;          /* discriminator value for the external CHOICE */
    Odr_fun fun;       /* decoder function */
} Z_ext_typeent;
```

The `what` member contains the `Z_External` union discriminator value for the given type: For the SUTRS record syntax, the value would be `Z_External_sutrs`. The `fun` member contains a pointer to the function which encodes/decodes the given type. Again, for the SUTRS record syntax, the value of `fun` would be `z_SUTRS` (a function pointer).

If you receive an EXTERNAL which contains an octet-string value that you suspect of being an ASN.1-structured data value, you can use `z_ext_gettypebyref` to look for the provided direct-reference. If the return value is different from NULL, you can use the provided function to decode the BER string (see section Using ODR).

If you want to *send* EXTERNALs containing ASN.1-structured values in the occtet-aligned branch of the CHOICE, this is possible too. However, on the encoding phase, it requires a somewhat involved juggling around of the various buffers involved.

If you need to add new, externally defined data types, you must update the struct above, in the source file `prt-ext.h`, as well as the encoder/decoder in the file `prt-ext.c`. When changing the latter, remember to update both the `arm` arrary and the list `type_table`, which drives the CHOICE biasing that is necessary to tell the different, structured types apart on decoding.

> **Note:** Eventually, the EXTERNAL processing will most likely automatically insert the correct OIDs or indirect-refs. First, however, we need to determine how application-context management (specifically the presentation-context-list) should fit into the various modules.

PDU Contents Table

We include, for reference, a listing of the fields of each top-level PDU, as well as their default settings.

Table 6-1. Default settings for PDU Initialize Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
protocolVersion	Odr_bitmask	Empty bitmask
options	Odr_bitmask	Empty bitmask
preferredMessageSize	int	30*1024
maximumRecordSize	int	30*1024
idAuthentication	Z_IdAuthentication	NULL
implementationId	char*	"81"
implementationName	char*	"YAZ"
implementationVersion	char*	YAZ_VERSION
userInformationField	Z_UserInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-2. Default settings for PDU Initialize Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
protocolVersion	Odr_bitmask	Empty bitmask
options	Odr_bitmask	Empty bitmask
preferredMessageSize	int	30*1024
maximumRecordSize	int	30*1024
result	bool_t	TRUE
implementationId	char*	"id)"

Field	Type	Default Value
implementationName	char*	"YAZ"
implementationVersion	char*	YAZ_VERSION
userInformationField	Z_UserInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-3. Default settings for PDU Search Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
smallSetUpperBound	int	0
largeSetLowerBound	int	1
mediumSetPresentNumber	int	0
replaceIndicator	bool_t	TRUE
resultSetName	char *	"default"
num_databaseNames	int	0
databaseNames	char **	NULL
smallSetElementSetNames	Z_ElementSetNames	NULL
mediumSetElementSetNames	Z_ElementSetNames	NULL
preferredRecordSyntax	Odr_oid	NULL
query	Z_Query	NULL
additionalSearchInfo	Z_OtherInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-4. Default settings for PDU Search Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
resultCount	int	0
numberOfRecordsReturned	int	0
nextResultSetPosition	int	0
searchStatus	bool_t	TRUE
resultSetStatus	int	NULL
presentStatus	int	NULL
records	Z_Records	NULL
additionalSearchInfo	Z_OtherInformation	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-5. Default settings for PDU Present Request

Field	Type	Default Value

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
resultSetId	char*	"default"
resultSetStartPoint	int	1
numberOfRecordsRequested	int	10
num_ranges	int	0
additionalRanges	Z_Range	NULL
recordComposition	Z_RecordComposition	NULL
preferredRecordSyntax	Odr_oid	NULL
maxSegmentCount	int	NULL
maxRecordSize	int	NULL
maxSegmentSize	int	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-6. Default settings for PDU Present Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
numberOfRecordsReturned	int	0
nextResultSetPosition	int	0
presentStatus	int	Z_PRES_SUCCESS
records	Z_Records	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-7. Default settings for Delete Result Set Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
deleteFunction	int	Z_DeleteRequest_list
num_ids	int	0
resultSetList	char**	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-8. Default settings for Delete Result Set Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
deleteOperationStatus	int	Z_DeleteStatus_success
num_statuses	int	0
deleteListStatuses	Z_ListStatus**	NULL
numberNotDeleted	int	NULL

Field	Type	Default Value
num_bulkStatuses	int	0
bulkStatuses	Z_ListStatus	NUL L
deleteMessage	char*	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-9. Default settings for Scan Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
num_databaseNames	int	0
databaseNames	char**	NULL
attributeSet	Odr_oid	NULL
termListAndStartPoint	Z_AttributesPlus...	NULL
stepSize	int	NULL
numberOfTermsRequested	int	20
preferredPositionInResponse	int	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-10. Default settings for Scan Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
stepSize	int	NULL
scanStatus	int	Z_Scan_success
numberOfEntriesReturned	int	0
positionOfTerm	int	NULL
entries	Z_ListEntris	NULL
attributeSet	Odr_oid	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-11. Default settings for Trigger Resource Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
requestedAction	int	Z_TriggerResourceCtrl_resou..
prefResourceReportFormat	Odr_oid	NULL
resultSetWanted	bool_t	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-12. Default settings for Resource Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
suspendedFlag	bool_t	NULL
resourceReport	Z_External	NULL
partialResultsAvailable	int	NULL
responseRequired	bool_t	FALSE
triggeredRequestFlag	bool_t	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-13. Default settings for Resource Control Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
continueFlag	bool_t	TRUE
resultSetWanted	bool_t	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-14. Default settings for Access Control Request

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
which	enum	Z_AccessRequest_simpleForm;
u	union	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-15. Default settings for Access Control Response

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
which	enum	Z_AccessResponse_simpleForm
u	union	NULL
diagnostic	Z_DiagRec	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-16. Default settings for Segment

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
numberOfRecordsReturned	int	value=0
num_segmentRecords	int	0

Field	Type	Default Value
segmentRecords	Z_NamePlusRecord	NULL
otherInfo	Z_OtherInformation	NULL

Table 6-17. Default settings for Close

Field	Type	Default Value
referenceId	Z_ReferenceId	NULL
closeReason	int	Z_Close_finished
diagnosticInformation	char*	NULL
resourceReportFormat	Odr_oid	NULL
resourceFormat	Z_External	NULL
otherInfo	Z_OtherInformation	NULL

Chapter 7. SOAP and SRW

Introduction

YAZ uses a very simple implementation of SOAP that only, currenly, supports what is sufficient to offer SRW functionality. The implementation uses the tree API (http://www.xmlsoft.org/html/libxml-tree.html) of libxml2 to encode and decode SOAP packages.

Like the Z39.50 ASN.1 module, the YAZ SRW implementation uses simple C structs to represent SOAP packages as well as HTTP packages.

HTTP

YAZ only offers HTTP as transport carrier for SOAP, but it is relatively easy to change that.

The following definition of `Z_GDU` (Generic Data Unit) allows for both HTTP and Z39.50 in one packet.

```
#include <yaz/zgdu.h>

#define Z_GDU_Z3950         1
#define Z_GDU_HTTP_Request  2
#define Z_GDU_HTTP_Response 3
typedef struct {
  int which;
  union {
    Z_APDU *z3950;
    Z_HTTP_Request *HTTP_Request;
    Z_HTTP_Response *HTTP_Response;
  } u;
} Z_GDU ;
```

The corresponding Z_GDU encoder/decoder is `z_GDU`. The `z3950` is any of the known BER encoded Z39.50 APDUs. `HTTP_Request` and `HTTP_Response` is the HTTP Request and Response respectively.

SOAP Packages

Every SOAP package in YAZ is represented as follows:

```
#include <yaz/soap.h>

typedef struct {
    char *fault_code;
    char *fault_string;
    char *details;
} Z_SOAP_Fault;

typedef struct {
    int no;
```

```
    char *ns;
    void *p;
} Z_SOAP_Generic;

#define Z_SOAP_fault 1
#define Z_SOAP_generic 2
#define Z_SOAP_error 3
typedef struct {
    int which;
    union {
        Z_SOAP_Fault   *fault;
        Z_SOAP_Generic *generic;
        Z_SOAP_Fault   *soap_error;
    } u;
    const char *ns;
} Z_SOAP;
```

The `fault` and `soap_error` arms represent both a SOAP fault - struct `Z_SOAP_Fault`. Any other generic (valid) package is represented by `Z_SOAP_Generic`.

The `ns` as part of `Z_SOAP` is the namespace for SOAP itself and reflects the SOAP version. For version 1.1 it is `http://schemas.xmlsoap.org/soap/envelope/`, for version 1.2 it is `http://www.w3.org/2001/06/soap-envelope`.

```
int z_soap_codec(ODR o, Z_SOAP **pp,
                 char **content_buf, int *content_len,
                 Z_SOAP_Handler *handlers);
```

The `content_buf` and `content_len` is XML buffer and length of buffer respectively.

The `handlers` is a list of SOAP codec handlers - one handler for each service namespace. For SRW, the namespace would be `http://www.loc.gov/zing/srw/v1.0/`.

When decoding, the `z_soap_codec` inspects the XML content and tries to match one of the services namespaces of the supplied handlers. If there is a match a handler function is invoked which decodes that particular SOAP package. If successful, the returned `Z_SOAP` package will be of type `Z_SOAP_Generic`. Member `no` is set the offset of handler that matched; `ns` is set to namespace of matching handler; the void pointer `p` is set to the C data structure assocatiated with the handler.

When a NULL namespace is met (member `ns` bwlow), that specifies end-of-list.

Each handler is defined as follows:

```
typedef struct {
    char *ns;
    void *client_data;
    Z_SOAP_fun f;
} Z_SOAP_Handler;
```

The ns is namespace of service associated with handler f. client_data is user-defined data which is passed to handler.

The prototype for a SOAP service handler is:

```
int handler(ODR o, void * ptr, void **handler_data,
            void *client_data, const char *ns);
```

The *o* specifies the mode (decode/encode) as usual. The second argument, *ptr*, is a libxml2 tree node pointer (xmlNodePtr) and is a pointer to the Body element of the SOAP package. The *handler_data* is an opaque pointer to a C definitions associated with the SOAP service. *client_data* is the pointer which was set as part of the Z_SOAP_handler. Finally, *ns* the service namespace.

SRW

SRW is just one implementation of a SOAP handler as described in the previous section. The encoder/decoder handler for SRW is defined as follows:

```
#include <yaz/srw.h>

int yaz_srw_codec(ODR o, void * pptr,
                  Z_SRW_GDU **handler_data,
                  void *client_data, const char *ns);
```

Here, Z_SRW_GDU is either searchRetrieverequest or a searchRetrieveResponse.

> **Note:** The xQuery and xSortKeys are not handled yet by the SRW implementation of YAZ. Explain is also missing. Future versions of YAZ will include these features.

The definition of searchRetrieveRequest is:

```
typedef struct {

#define Z_SRW_query_type_cql  1
#define Z_SRW_query_type_xcql 2
#define Z_SRW_query_type_pqf  3
    int query_type;
    union {
        char *cql;
        char *xcql;
        char *pqf;
    } query;

#define Z_SRW_sort_type_none 1
#define Z_SRW_sort_type_sort 2
#define Z_SRW_sort_type_xSort 3
    int sort_type;
    union {
```

```
        char *none;
        char *sortKeys;
        char *xSortKeys;
    } sort;
    int  *startRecord;
    int  *maximumRecords;
    char *recordSchema;
    char *recordPacking;
    char *database;
} Z_SRW_searchRetrieveRequest;
```

Please observe that data of type xsd:string is represented as a char pointer (`char *`). A null pointer means that the element is absent. Data of type xsd:integer is representd as a pointer to an int (`int *`). Again, a null pointer us used for absent elements.

The SearchRetrieveResponse has the following definition.

```
typedef struct {
    int * numberOfRecords;
    char * resultSetId;
    int * resultSetIdleTime;

    Z_SRW_record *records;
    int num_records;

    Z_SRW_diagnostic *diagnostics;
    int num_diagnostics;
    int *nextRecordPosition;
} Z_SRW_searchRetrieveResponse;
```

The `num_records` and `num_diagnostics` is number of returned records and diagnostics respectively and also correspond to the "size of" arrays `records` and `diagnostics`.

A retrieval record is defined as follows:

```
typedef struct {
    char *recordSchema;
    char *recordData_buf;
    int recordData_len;
    int *recordPosition;
} Z_SRW_record;
```

The record data is defined as a buffer of some length so that data can be of any type. SRW 1.0 currenly doesn’t allow for this (only XML), but future versions might do.

And, a diagnostic as:

```
typedef struct {
    int  *code;
    char *details;
} Z_SRW_diagnostic;
```

Chapter 8. Supporting Tools

In support of the service API - primarily the ASN module, which provides the pro-grammatic interface to the Z39.50 APDUs, YAZ contains a collection of tools that support the development of applications.

Query Syntax Parsers

Since the type-1 (RPN) query structure has no direct, useful string representation, every origin application needs to provide some form of mapping from a local query notation or representation to a Z_RPNQuery structure. Some programmers will prefer to construct the query manually, perhaps using `odr_malloc()` to simplify memory management. The YAZ distribution includes two separate, query-generating tools that may be of use to you.

Prefix Query Format

Since RPN or reverse polish notation is really just a fancy way of describing a suffix notation format (operator follows operands), it would seem that the confusion is total when we now introduce a prefix notation for RPN. The reason is one of simple laziness - it's somewhat simpler to interpret a prefix format, and this utility was designed for maximum simplicity, to provide a baseline representation for use in simple test applications and scripting environments (like Tcl). The demonstration client included with YAZ uses the PQF.

> **Note:** The PQF have been adopted by other parties developing Z39.50 software. It is often referred to as Prefix Query Notation - PQN.

The PQF is defined by the pquery module in the YAZ library. There are two sets of function that have similar behavior. First set operates on a PQF parser handle, second set doesn't. First set set of functions are more flexible than the second set. Second set is obsolete and is only provided to ensure backwards compatibility.

First set of functions all operate on a PQF parser handle:

```
#include <yaz/pquery.h>

YAZ_PQF_Parser yaz_pqf_create (void);

void yaz_pqf_destroy (YAZ_PQF_Parser p);

Z_RPNQuery *yaz_pqf_parse (YAZ_PQF_Parser p, ODR o, const char *qbuf);

Z_AttributesPlusTerm *yaz_pqf_scan (YAZ_PQF_Parser p, ODR o,
                    Odr_oid **attributeSetId, const char *qbuf);


int yaz_pqf_error (YAZ_PQF_Parser p, const char **msg, size_t *off);
```

A PQF parser is created and destructed by functions `yaz_pqf_create` and `yaz_pqf_destroy` respectively. Function `yaz_pqf_parse` parses query given by string `qbuf`. If parsing was successful, a Z39.50 RPN Query is returned which is created using ODR stream `o`. If parsing failed, a NULL pointer is returned. Function `yaz_pqf_scan` takes a scan query in `qbuf`. If parsing was successful, the function returns attributes plus term pointer and modifies `attributeSetId` to hold attribute set for the scan request - both allocated using ODR stream `o`. If parsing failed, yaz_pqf_scan returns a NULL pointer. Error information for bad queries can be obtained by a call to `yaz_pqf_error` which returns an error code and modifies `*msg` to point to an error description, and modifies `*off` to the offset within last query were parsing failed.

The second set of functions are declared as follows:

```
#include <yaz/pquery.h>

Z_RPNQuery *p_query_rpn (ODR o, oid_proto proto, const char *qbuf);

Z_AttributesPlusTerm *p_query_scan (ODR o, oid_proto proto,
                        Odr_oid **attributeSetP, const char *qbuf);

int p_query_attset (const char *arg);
```

The function `p_query_rpn()` takes as arguments an ODR stream (see section The ODR Module) to provide a memory source (the structure created is released on the next call to `odr_reset()` on the stream), a protocol identifier (one of the constants PROTO_Z3950 and PROTO_SR), an attribute set reference, and finally a null-terminated string holding the query string.

If the parse went well, `p_query_rpn()` returns a pointer to a `Z_RPNQuery` structure which can be placed directly into a `Z_SearchRequest`. If parsing failed, due to syntax error, a NULL pointer is returned.

The `p_query_attset` specifies which attribute set to use if the query doesn't specify one by the `@attrset` operator. The `p_query_attset` returns 0 if the argument is a valid attribute set specifier; otherwise the function returns -1.

The grammar of the PQF is as follows:

query ::= top-set query-struct.

top-set ::= ['@attrset' string]

query-struct ::= attr-spec | simple | complex | '@term' term-type

attr-spec ::= '@attr' [string] string query-struct

complex ::= operator query-struct query-struct.

operator ::= '@and' | '@or' | '@not' | '@prox' proximity.

simple ::= result-set | term.

result-set ::= '@set' string.

term ::= string.

proximity ::= exclusion distance ordered relation which-code unit-code.

exclusion ::= ’1’ | ’0’ | ’void’.

distance ::= integer.

ordered ::= ’1’ | ’0’.

relation ::= integer.

which-code ::= ’known’ | ’private’ | integer.

unit-code ::= integer.

term-type ::= ’general’ | ’numeric’ | ’string’ | ’oid’ | ’datetime’ | ’null’.

You will note that the syntax above is a fairly faithful representation of RPN, except for the Attribute, which has been moved a step away from the term, allowing you to associate one or more attributes with an entire query structure. The parser will automatically apply the given attributes to each term as required.

The @attr operator is followed by an attribute specification (`attr-spec` above). The specification consists of optional an attribute set, an attribute type-value pair and a sub query. The attribute type-value pair is packed in one string: an attribute type, a dash, followed by an attribute value. The type is always an integer but the value may be either an integer or a string (if it doesn’t start with a digit character).

Version 3 of the Z39.50 specification defines various encoding of terms. Use the `@term` *`type`*, where type is one of: `general`, `numeric`, `string` (for InternationalString), .. If no term type has been given, the `general` form is used which is the only encoding allowed in both version 2 - and 3 of the Z39.50 standard.

Example 8-1. PQF queries

Queries using simple terms.

```
dylan
"bob dylan"
```

Boolean operators.

```
@or "dylan" "zimmerman"
@and @or dylan zimmerman when
@and when @or dylan zimmerman
```

Reference to result sets.

```
@set Result-1
```

```
@and @set seta setb
```

Attributes for terms.

```
@attr 1=4 computer
@attr 1=4 @attr 4=1 "self portrait"
@attr exp1 @attr 1=1 CategoryList
@attr gils 1=2008 Copenhagen
@attr 1=/book/title computer
```

Proximity.

```
@prox 0 3 1 2 k 2 dylan zimmerman
```

Specifying term type.

```
@term string "a UTF-8 string, maybe?"
```

Mixed queries

```
@or @and bob dylan @set Result-1

@attr 4=1 @and @attr 1=1 "bob dylan" @attr 1=4 "slow train coming"

@and @attr 2=4 @attr gils 1=2038 -114 @attr 2=2 @attr gils 1=2039 -109
```

Common Command Language

Not all users enjoy typing in prefix query structures and numerical attribute values, even in a minimalistic test client. In the library world, the more intuitive Common Command Language (or ISO 8777) has enjoyed some popularity - especially before the widespread availability of graphical interfaces. It is still useful in applications where you for some reason or other need to provide a symbolic language for expressing boolean query structures.

The EUROPAGATE (http://europagate.dtv.dk/) research project working under the Libraries programme of the European Commission's DG XIII has, amongst other useful tools, implemented a general-purpose CCL parser which produces an output structure that can be trivially converted to the internal RPN representation of YAZ (The `Z_RPNQuery` structure). Since the CCL utility - along with the rest of the software produced by EUROPAGATE - is made freely available on a liberal license, it is included as a supplement to YAZ.

CCL Syntax

The CCL parser obeys the following grammar for the FIND argument. The syntax is annotated by in the lines prefixed by `--`.

```
CCL-Find ::= CCL-Find Op Elements
           | Elements.
```

```
Op ::= "and" | "or" | "not"
-- The above means that Elements are separated by boolean operators.

Elements ::= '(' CCL-Find ')'
          | Set
          | Terms
          | Qualifiers Relation Terms
          | Qualifiers Relation '(' CCL-Find ')'
          | Qualifiers '=' string '-' string
-- Elements is either a recursive definition, a result set reference, a
-- list of terms, qualifiers followed by terms, qualifiers followed
-- by a recursive definition or qualifiers in a range (lower - upper).

Set ::= 'set' = string
-- Reference to a result set

Terms ::= Terms Prox Term
       | Term
-- Proximity of terms.

Term ::= Term string
      | string
-- This basically means that a term may include a blank

Qualifiers ::= Qualifiers ',' string
            | string
-- Qualifiers is a list of strings separated by comma

Relation ::= '=' | '>=' | '<=' | '<>' | '>' | '<'
-- Relational operators. This really doesn't follow the ISO8777
-- standard.

Prox ::= '%' | '!'
-- Proximity operator
```

The following queries are all valid:

```
dylan

"bob dylan"

dylan or zimmerman

set=1

(dylan and bob) or set=1
```

Assuming that the qualifiers `ti`, `au` and `date` are defined we may use:

```
ti=self portrait

au=(bob dylan and slow train coming)

date>1980 and (ti=((self portrait)))
```

CCL Qualifiers

Qualifiers are used to direct the search to a particular searchable index, such as title (ti) and author indexes (au). The CCL standard itself doesn't specify a particular set of qualifiers, but it does suggest a few short-hand notations. You can customize the CCL parser to support a particular set of qualifiers to reflect the current target profile. Traditionally, a qualifier would map to a particular use-attribute within the BIB-1 attribute set. However, you could also define qualifiers that would set, for example, the structure-attribute.

Consider a scenario where the target support ranked searches in the title-index. In this case, the user could specify

```
ti,ranked=knuth computer
```

and the `ranked` would map to relation=relevance (2=102) and the `ti` would map to title (1=4).

A "profile" with a set predefined CCL qualifiers can be read from a file. The YAZ client reads its CCL qualifiers from a file named `default.bib`. Each line in the file has the form:

`qualifier-name` *`type`*=*`val`* *`type`*=*`val`* ...

where *`qualifier-name`* is the name of the qualifier to be used (eg. `ti`), *`type`* is a BIB-1 category type and *`val`* is the corresponding BIB-1 attribute value. The *`type`* can be either numeric or it may be either `u` (use), `r` (relation), `p` (position), `s` (structure), `t` (truncation) or `c` (completeness). The *`qualifier-name`* `term` has a special meaning. The types and values for this definition is used when *no* qualifiers are present.

Consider the following definition:

```
ti       u=4 s=1
au       u=1 s=1
term     s=105
```

Two qualifiers are defined, `ti` and `au`. They both set the structure-attribute to phrase (1). `ti` sets the use-attribute to 4. `au` sets the use-attribute to 1. When no qualifiers are used in the query the structure-attribute is set to free-form-text (105).

CCL API

All public definitions can be found in the header file `ccl.h`. A profile identifier is of type `CCL_bibset`. A profile must be created with the call to the function `ccl_qual_mk` which returns a profile handle of type `CCL_bibset`.

To read a file containing qualifier definitions the function `ccl_qual_file` may be convenient. This function takes an already opened `FILE` handle pointer as argument along with a `CCL_bibset` handle.

To parse a simple string with a FIND query use the function

```
struct ccl_rpn_node *ccl_find_str (CCL_bibset bibset, const char *str,
                                   int *error, int *pos);
```

which takes the CCL profile (`bibset`) and query (`str`) as input. Upon successful completion the RPN tree is returned. If an error occur, such as a syntax error, the integer pointed to by `error` holds the error code and `pos` holds the offset inside query string in which the parsing failed.

An English representation of the error may be obtained by calling the `ccl_err_msg` function. The error codes are listed in `ccl.h`.

To convert the CCL RPN tree (type `struct ccl_rpn_node *`) to the Z_RPNQuery of YAZ the function `ccl_rpn_query` must be used. This function which is part of YAZ is implemented in `yaz-ccl.c`. After calling this function the CCL RPN tree is probably no longer needed. The `ccl_rpn_delete` destroys the CCL RPN tree.

A CCL profile may be destroyed by calling the `ccl_qual_rm` function.

The token names for the CCL operators may be changed by setting the globals (all type `char *`) `ccl_token_and`, `ccl_token_or`, `ccl_token_not` and `ccl_token_set`. An operator may have aliases, i.e. there may be more than one name for the operator. To do this, separate each alias with a space character.

CQL

CQL (http://www.loc.gov/z3950/agency/zing/cql/) - Common Query Language - was defined for the SRW (http://www.loc.gov/z3950/agency/zing/srw/) protocol. In many ways CQL has a similar syntax to CCL. The objective of CQL is different. Where CCL aims to be an end-user language, CQL is *the* protocol query language for SRW.

> **Tip:** If you are new to CQL, read the Gentle Introduction (http://zing.z3950.org/cql/intro.html).

The CQL parser in YAZ provides the following:

- It parses and validates a CQL query.
- It generates a C structure that allows you to convert a CQL query to some other query language, such as SQL.
- The parser converts a valid CQL query to PQF, thus providing a way to use CQL for both SRW/SRU servers and Z39.50 targets at the same time.
- The parser converts CQL to XCQL (http://www.loc.gov/z3950/agency/zing/cql/xcql.html). XCQL is an XML representation of CQL. XCQL is part of the SRW specification. However, since SRU

supports CQL only, we don't expect XCQL to be widely used. Furthermore, CQL has the advantage over XCQL that it is easy to read.

CQL parsing

A CQL parser is represented by the `CQL_parser` handle. Its contents should be considered YAZ internal (private).

```
#include <yaz/cql.h>

typedef struct cql_parser *CQL_parser;

CQL_parser cql_parser_create(void);
void cql_parser_destroy(CQL_parser cp);
```

A parser is created by `cql_parser_create` and is destroyed by `cql_parser_destroy`.

To parse a CQL query string, the following function is provided:

```
int cql_parser_string(CQL_parser cp, const char *str);
```

A CQL query is parsed by the `cql_parser_string` which takes a query *`str`*. If the query was valid (no syntax errors), then zero is returned; otherwise a non-zero error code is returned.

```
int cql_parser_stream(CQL_parser cp,
                      int (*getbyte)(void *client_data),
                      void (*ungetbyte)(int b, void *client_data),
                      void *client_data);

int cql_parser_stdio(CQL_parser cp, FILE *f);
```

The functions `cql_parser_stream` and `cql_parser_stdio` parses a CQL query - just like `cql_parser_string`. The only difference is that the CQL query can be fed to the parser in different ways. The `cql_parser_stream` uses a generic byte stream as input. The `cql_parser_stdio` uses a `FILE` handle which is opened for reading.

CQL tree

The the query string is validl, the CQL parser generates a tree representing the structure of the CQL query.

```
struct cql_node *cql_parser_result(CQL_parser cp);
```

`cql_parser_result` returns the a pointer to the root node of the resulting tree.

Each node in a CQL tree is represented by a `struct cql_node`. It is defined as follows:

```
#define CQL_NODE_ST 1
#define CQL_NODE_BOOL 2
#define CQL_NODE_MOD 3
struct cql_node {
    int which;
    union {
        struct {
            char *index;
            char *term;
            char *relation;
            struct cql_node *modifiers;
            struct cql_node *prefixes;
        } st;
        struct {
            char *value;
            struct cql_node *left;
            struct cql_node *right;
            struct cql_node *modifiers;
            struct cql_node *prefixes;
        } boolean;
        struct {
            char *name;
            char *value;
            struct cql_node *next;
        } mod;
    } u;
};
```

There are three kinds of nodes, search term (ST), boolean (BOOL), and modifier (MOD).

The search term node has five members:

- `index`: index for search term. If an index is unspecified for a search term, `index` will be NULL.
- `term`: the search term itself.
- `relation`: relation for search term.
- `modifiers`: relation modifiers for search term. The `modifiers` is a simple linked list (NULL for last entry). Each relation modifier node is of type `MOD`.
- `prefixes`: index prefixes for search term. The `prefixes` is a simple linked list (NULL for last entry). Each prefix node is of type `MOD`.

The boolean node represents both `and`, `or`, not as well as proximity.

- `left` and `right`: left - and right operand respectively.
- `modifiers`: proximity arguments.
- `prefixes`: index prefixes. The `prefixes` is a simple linked list (NULL for last entry). Each prefix node is of type `MOD`.

The modifier node is a "utility" node used for name-value pairs, such as prefixes, proximity arguements, etc.

- `name` name of mod node.
- `value` value of mod node.
- `next`: pointer to next node which is always a mod node (NULL for last entry).

CQL to PQF conversion

Conversion to PQF (and Z39.50 RPN) is tricky by the fact that the resulting RPN depends on the Z39.50 target capabilities (combinations of supported attributes). In addition, the CQL and SRW operates on index prefixes (URI or strings), whereas the RPN uses Object Identifiers for attribute sets.

The CQL library of YAZ defines a `cql_transform_t` type. It represents a particular mapping between CQL and RPN. This handle is created and destroyed by the functions:

```
cql_transform_t cql_transform_open_FILE (FILE *f);
cql_transform_t cql_transform_open_fname(const char *fname);
void cql_transform_close(cql_transform_t ct);
```

The first two functions create a tranformation handle from either an already open FILE or from a filename respectively.

The handle is destroyed by `cql_transform_close` in which case no further reference of the handle is allowed.

When a `cql_transform_t` handle has been created you can convert to RPN.

```
int cql_transform_buf(cql_transform_t ct,
                      struct cql_node *cn, char *out, int max);
```

This function converts the CQL tree `cn` using handle `ct`. For the resulting PQF, you supply a buffer `out` which must be able to hold at at least `max` characters.

If conversion failed, `cql_transform_buf` returns a non-zero error code; otherwise zero is returned (conversion successful).

If you wish to be able to produce a PQF result in a different way, there are two alternatives.

```
void cql_transform_pr(cql_transform_t ct,
                      struct cql_node *cn,
                      void (*pr)(const char *buf, void *client_data),
                      void *client_data);

int cql_transform_FILE(cql_transform_t ct,
                       struct cql_node *cn, FILE *f);
```

The former function produces output to a user-defined output stream. The latter writes the result to an already open `FILE`.

Specification of CQL to RPN mapping

The file supplied to functions `cql_transform_open_FILE`, `cql_transform_open_fname` follows a structure found in many Unix utilities. It consists of mapping specifications - one per line. Lines starting with # are ignored (comments).

Each line is of the form

`CQL pattern = RPN equivalent`

An RPN pattern is a simple attribute list. Each attribute pair takes the form:

[*`set`*] *`type=value`*

The attribute *`set`* is optional. The *`type`* is the attribute type, *`value`* the attribute value.

The following CQL patterns are recognized:

`qualifier.`*`set.name`*

This pattern is invoked when a CQL qualifier, such as dc.title is converted. *`set`* and *`name`* is the index set and qualifier name respectively. Typically, the RPN specifies an equivalent use attribute.

For terms not bound by a qualifier the pattern `qualifier.srw.serverChoice` is used. Here, the prefix `srw` is defined as `http://www.loc.gov/zing/cql/srw-indexes/v1.0/`. If this pattern is not defined, the mapping will fail.

`relation.`*`relation`*

This pattern specifies how a CQL relation is mapped to RPN. *`pattern`* is name of relation operator. Since = is used as separator between CQL pattern and RPN, CQL relations including = cannot be used directly. To avoid a conflict, the names `ge`, `eq`, `le`, must be used for CQL operators, greater-than-or-equal, equal, less-than-or-equal respectively. The RPN pattern is supposed to include a relation attribute.

For terms not bound by a relation, the pattern `relation.scr` is used. If the pattern is not defined, the mapping will fail.

The special pattern, `relation.*` is used when no other relation pattern is matched.

`relationModifier.`*`mod`*

This pattern specifies how a CQL relation modifier is mapped to RPN. The RPN pattern is usually a relation attribute.

`structure.`*`type`*

This pattern specifies how a CQL structure is mapped to RPN. Note that this CQL pattern is somewhat to similar to CQL pattern `relation`. The *`type`* is a CQL relation.

The pattern, `structure.*` is used when no other structure pattern is matched. Usually, the RPN equivalent specifies a structure attribute.

`position.`*`type`*

This pattern specifies how the anchor (position) of CQL is mapped to RPN. The *`type`* is one of `first`, `any`, `last`, `firstAndLast`.

The pattern, `position.*` is used when no other position pattern is matched.

`set.`*`prefix`*

This specification defines a CQL index set for a given prefix. The value on the right hand side is the URI for the set - *not* RPN. All prefixes used in qualifier patterns must be defined this way.

Example 8-2. Small CQL to RPN mapping file

This small file defines two index sets, three qualifiers and three relations, a position pattern and a default structure.

```
set.srw    = http://www.loc.gov/zing/cql/srw-indexes/v1.0/
set.dc     = http://www.loc.gov/zing/cql/dc-indexes/v1.0/

qualifier.srw.serverChoice = 1=1016
qualifier.dc.title         = 1=4
qualifier.dc.subject       = 1=21

relation.<                 = 2=1
relation.eq                = 2=3
relation.scr               = 2=3

position.any               = 3=3 6=1

structure.*                = 4=1
```

With the mappings above, the CQL query

```
computer
```

is converted to the PQF:

```
@attr 1=1016 @attr 2=3 @attr 4=1 @attr 3=3 @attr 6=1 "computer"
```

by rules `qualifier.srw.serverChoice`, `relation.scr`, `structure.*`, `position.any`.

CQL query

```
computer^
```

is rejected, since `position.right` is undefined.

CQL query

```
>my = "http://www.loc.gov/zing/cql/dc-indexes/v1.0/" my.title = x
```

is converted to

```
@attr 1=4 @attr 2=3 @attr 4=1 @attr 3=3 @attr 6=1 "x"
```

CQL to XCQL conversion

Conversion from CQL to XCQL is trivial and does not require a mapping to be defined. There three functions to choose from depending on the way you wish to store the resulting output (XML buffer containing XCQL).

```
int cql_to_xml_buf(struct cql_node *cn, char *out, int max);
void cql_to_xml(struct cql_node *cn,
                void (*pr)(const char *buf, void *client_data),
                void *client_data);
void cql_to_xml_stdio(struct cql_node *cn, FILE *f);
```

Function `cql_to_xml_buf` converts to XCQL and stores result in a user supplied buffer of a given max size.

`cql_to_xml` writes the result in a user defined output stream. `cql_to_xml_stdio` writes to a a file.

Object Identifiers

The basic YAZ representation of an OID is an array of integers, terminated with the value -1. The ODR module provides two utility-functions to create and copy this type of data elements:

```
Odr_oid *odr_getoidbystr(ODR o, char *str);
```

Creates an OID based on a string-based representation using dots (.) to separate elements in the OID.

```
Odr_oid *odr_oiddup(ODR odr, Odr_oid *o);
```

Creates a copy of the OID referenced by the *o* parameter. Both functions take an ODR stream as parameter. This stream is used to allocate memory for the data elements, which is released on a subsequent call to `odr_reset()` on that stream.

The OID module provides a higher-level representation of the family of object identifiers which describe the Z39.50 protocol and its related objects. The definition of the module interface is given in the `oid.h` file.

The interface is mainly based on the `oident` structure. The definition of this structure looks like this:

```
typedef struct oident
{
    oid_proto proto;
    oid_class oclass;
    oid_value value;
    int oidsuffix[OID_SIZE];
    char *desc;
} oident;
```

The proto field takes one of the values

```
PROTO_Z3950
PROTO_SR
```

If you don't care about talking to SR-based implementations (few exist, and they may become fewer still if and when the ISO SR and ANSI Z39.50 documents are merged into a single standard), you can ignore this field on incoming packages, and always set it to PROTO_Z3950 for outgoing packages.

The oclass field takes one of the values

```
CLASS_APPCTX
CLASS_ABSYN
CLASS_ATTSET
CLASS_TRANSYN
CLASS_DIAGSET
CLASS_RECSYN
CLASS_RESFORM
CLASS_ACCFORM
CLASS_EXTSERV
CLASS_USERINFO
CLASS_ELEMSPEC
CLASS_VARSET
CLASS_SCHEMA
CLASS_TAGSET
CLASS_GENERAL
```

corresponding to the OID classes defined by the Z39.50 standard. Finally, the value field takes one of the values

```
VAL_APDU
VAL_BER
VAL_BASIC_CTX
VAL_BIB1
VAL_EXP1
VAL_EXT1
```

```
VAL_CCL1
VAL_GILS
VAL_WAIS
VAL_STAS
VAL_DIAG1
VAL_ISO2709
VAL_UNIMARC
VAL_INTERMARC
VAL_CCF
VAL_USMARC
VAL_UKMARC
VAL_NORMARC
VAL_LIBRISMARC
VAL_DANMARC
VAL_FINMARC
VAL_MAB
VAL_CANMARC
VAL_SBN
VAL_PICAMARC
VAL_AUSMARC
VAL_IBERMARC
VAL_EXPLAIN
VAL_SUTRS
VAL_OPAC
VAL_SUMMARY
VAL_GRS0
VAL_GRS1
VAL_EXTENDED
VAL_RESOURCE1
VAL_RESOURCE2
VAL_PROMPT1
VAL_DES1
VAL_KRB1
VAL_PRESSET
VAL_PQUERY
VAL_PCQUERY
VAL_ITEMORDER
VAL_DBUPDATE
VAL_EXPORTSPEC
VAL_EXPORTINV
VAL_NONE
VAL_SETM
VAL_SETG
VAL_VAR1
VAL_ESPEC1
```

again, corresponding to the specific OIDs defined by the standard.

The desc field contains a brief, mnemonic name for the OID in question.

The function

```
struct oident *oid_getentbyoid(int *o);
```

takes as argument an OID, and returns a pointer to a static area containing an `oident` structure. You typically use this function when you receive a PDU containing an OID, and you wish to branch out depending on the specific OID value.

The function

```
int *oid_ent_to_oid(struct oident *ent, int *dst);
```

Takes as argument an `oident` structure - in which the `proto`, `oclass`/, and `value` fields are assumed to be set correctly - and returns a pointer to a the buffer as given by `dst` containing the base representation of the corresponding OID. The function returns NULL and the array dst is unchanged if a mapping couldn't place. The array `dst` should be at least of size `OID_SIZE`.

The `oid_ent_to_oid()` function can be used whenever you need to prepare a PDU containing one or more OIDs. The separation of the `protocol` element from the remainder of the OID-description makes it simple to write applications that can communicate with either Z39.50 or OSI SR-based applications.

The function

```
oid_value oid_getvalbyname(const char *name);
```

takes as argument a mnemonic OID name, and returns the `/value` field of the first entry in the database that contains the given name in its `desc` field.

Finally, the module provides the following utility functions, whose meaning should be obvious:

```
void oid_oidcpy(int *t, int *s);
void oid_oidcat(int *t, int *s);
int oid_oidcmp(int *o1, int *o2);
int oid_oidlen(int *o);
```

Note: The OID module has been criticized - and perhaps rightly so - for needlessly abstracting the representation of OIDs. Other toolkits use a simple string-representation of OIDs with good results. In practice, we have found the interface comfortable and quick to work with, and it is a simple matter (for what it's worth) to create applications compatible with both ISO SR and Z39.50. Finally, the use of the `/oident` database is by no means mandatory. You can easily create your own system for representing OIDs, as long as it is compatible with the low-level integer-array representation of the ODR module.

Nibble Memory

Sometimes when you need to allocate and construct a large, interconnected complex of structures, it can be a bit of a pain to release the associated memory again. For the structures describing the Z39.50 PDUs and related structures, it is convenient to use the memory-management system of the ODR subsystem (see Using ODR). However, in some circumstances where you might otherwise benefit from using a

simple nibble memory management system, it may be impractical to use `odr_malloc()` and `odr_reset()`. For this purpose, the memory manager which also supports the ODR streams is made available in the NMEM module. The external interface to this module is given in the `nmem.h` file.

The following prototypes are given:

```
NMEM nmem_create(void);
void nmem_destroy(NMEM n);
void *nmem_malloc(NMEM n, int size);
void nmem_reset(NMEM n);
int nmem_total(NMEM n);
void nmem_init(void);
void nmem_exit(void);
```

The `nmem_create()` function returns a pointer to a memory control handle, which can be released again by `nmem_destroy()` when no longer needed. The function `nmem_malloc()` allocates a block of memory of the requested size. A call to `nmem_reset()` or `nmem_destroy()` will release all memory allocated on the handle since it was created (or since the last call to `nmem_reset()`. The function `nmem_total()` returns the number of bytes currently allocated on the handle.

The nibble memory pool is shared amongst threads. POSIX mutex'es and WIN32 Critical sections are introduced to keep the module thread safe. Function `nmem_init()` initializes the nibble memory library and it is called automatically the first time the `YAZ.DLL` is loaded. YAZ uses function `DllMain` to achieve this. You should *not* call `nmem_init` or `nmem_exit` unless you're absolute sure what you're doing. Note that in previous YAZ versions you'd have to call `nmem_init` yourself.

Chapter 9. The ODR Module

Introduction

ODR is the BER-encoding/decoding subsystem of YAZ. Care as been taken to isolate ODR from the rest of the package - specifically from the transport interface. ODR may be used in any context where basic ASN.1/BER representations are used.

If you are only interested in writing a Z39.50 implementation based on the PDUs that are already provided with YAZ, you only need to concern yourself with the section on managing ODR streams (section Using ODR). Only if you need to implement ASN.1 beyond that which has been provided, should you worry about the second half of the documentation (section Programming with ODR). If you use one of the higher-level interfaces, you can skip this section entirely.

This is important, so we'll repeat it for emphasis: *You do not need to read section Programming with ODR to implement Z39.50 with YAZ.*

If you need a part of the protocol that isn't already in YAZ, you should contact the authors before going to work on it yourself: We might already be working on it. Conversely, if you implement a useful part of the protocol before us, we'd be happy to include it in a future release.

Using ODR

ODR Streams

Conceptually, the ODR stream is the source of encoded data in the decoding mode; when encoding, it is the receptacle for the encoded data. Before you can use an ODR stream it must be allocated. This is done with the function

```
ODR odr_createmem(int direction);
```

The `odr_createmem()` function takes as argument one of three manifest constants: `ODR_ENCODE`, `ODR_DECODE`, or `ODR_PRINT`. An ODR stream can be in only one mode - it is not possible to change its mode once it's selected. Typically, your program will allocate at least two ODR streams - one for decoding, and one for encoding.

When you're done with the stream, you can use

```
void odr_destroy(ODR o);
```

to release the resources allocated for the stream.

Memory Management

Two forms of memory management take place in the ODR system. The first one, which has to do with allocating little bits of memory (sometimes quite large bits of memory, actually) when a protocol

package is decoded, and turned into a complex of interlinked structures. This section deals with this system, and how you can use it for your own purposes. The next section deals with the memory management which is required when encoding data - to make sure that a large enough buffer is available to hold the fully encoded PDU.

The ODR module has its own memory management system, which is used whenever memory is required. Specifically, it is used to allocate space for data when decoding incoming PDUs. You can use the memory system for your own purposes, by using the function

```
void *odr_malloc(ODR o, int size);
```

You can't use the normal `free(2)` routine to free memory allocated by this function, and ODR doesn't provide a parallel function. Instead, you can call

```
void odr_reset(ODR o, int size);
```

when you are done with the memory: Everything allocated since the last call to `odr_reset()` is released. The `odr_reset()` call is also required to clear up an error condition on a stream.

The function

```
int odr_total(ODR o);
```

returns the number of bytes allocated on the stream since the last call to `odr_reset()`.

The memory subsystem of ODR is fairly efficient at allocating and releasing little bits of memory. Rather than managing the individual, small bits of space, the system maintains a free-list of larger chunks of memory, which are handed out in small bits. This scheme is generally known as a *nibble memory* system. It is very useful for maintaining short-lived constructions such as protocol PDUs.

If you want to retain a bit of memory beyond the next call to `odr_reset()`, you can use the function

```
ODR_MEM odr_extract_mem(ODR o);
```

This function will give you control of the memory recently allocated on the ODR stream. The memory will live (past calls to `odr_reset()`), until you call the function

```
void odr_release_mem(ODR_MEM p);
```

The opaque `ODR_MEM` handle has no other purpose than referencing the memory block for you until you want to release it.

You can use `odr_extract_mem()` repeatedly between allocating data, to retain individual control of separate chunks of data.

Encoding and Decoding Data

When encoding data, the ODR stream will write the encoded octet string in an internal buffer. To retrieve the data, use the function

```
char *odr_getbuf(ODR o, int *len, int *size);
```

The integer pointed to by len is set to the length of the encoded data, and a pointer to that data is returned. `*size` is set to the size of the buffer (unless `size` is null, signaling that you are not interested in the size). The next call to a primitive function using the same ODR stream will overwrite the data, unless a different buffer has been supplied using the call

```
void odr_setbuf(ODR o, char *buf, int len, int can_grow);
```

which sets the encoding (or decoding) buffer used by `o` to `buf`, using the length `len`. Before a call to an encoding function, you can use `odr_setbuf()` to provide the stream with an encoding buffer of sufficient size (length). The `can_grow` parameter tells the encoding ODR stream whether it is allowed to use `realloc(2)` to increase the size of the buffer when necessary. The default condition of a new encoding stream is equivalent to the results of calling

```
odr_setbuf(stream, 0, 0, 1);
```

In this case, the stream will allocate and reallocate memory as necessary. The stream reallocates memory by repeatedly doubling the size of the buffer - the result is that the buffer will typically reach its maximum, working size with only a small number of reallocation operations. The memory is freed by the stream when the latter is destroyed, unless it was assigned by the user with the `can_grow` parameter set to zero (in this case, you are expected to retain control of the memory yourself).

To assume full control of an encoded buffer, you must first call `odr_getbuf()` to fetch the buffer and its length. Next, you should call `odr_setbuf()` to provide a different buffer (or a null pointer) to the stream. In the simplest case, you will reuse the same buffer over and over again, and you will just need to call `odr_getbuf()` after each encoding operation to get the length and address of the buffer. Note that the stream may reallocate the buffer during an encoding operation, so it is necessary to retrieve the correct address after each encoding operation.

It is important to realize that the ODR stream will not release this memory when you call `odr_reset()`: It will merely update its internal pointers to prepare for the encoding of a new data value. When the stream is released by the `odr_destroy()` function, the memory given to it by `odr_setbuf` will be released *only* if the `can_grow` parameter to `odr_setbuf()` was nonzero. The `can_grow` parameter, in other words, is a way of signaling who is to own the buffer, you or the ODR stream. If you never call `odr_setbuf()` on your encoding stream, which is typically the case, the buffer allocated by the stream will belong to the stream by default.

When you wish to decode data, you should first call `odr_setbuf()`, to tell the decoding stream where to find the encoded data, and how long the buffer is (the `can_grow` parameter is ignored by a decoding stream). After this, you can call the function corresponding to the data you wish to decode (eg, `odr_integer()` odr `z_APDU()`).

Examples of encoding/decoding functions:

```
int odr_integer(ODR o, int **p, int optional, const char *name);

int z_APDU(ODR o, Z_APDU **p, int optional, const char *name);
```

If the data is absent (or doesn't match the tag corresponding to the type), the return value will be either 0 or 1 depending on the `optional` flag. If `optional` is 0 and the data is absent, an error flag will be raised in the stream, and you'll need to call `odr_reset()` before you can use the stream again. If `optional` is nonzero, the pointer *pointed* to/ by `p` will be set to the null value, and the function will return 1. The `name` argument is used to pretty-print the tag in question. It may be set to `NULL` if pretty-printing is not desired.

If the data value is found where it's expected, the pointer *pointed to* by the `p` argument will be set to point to the decoded type. The space for the type will be allocated and owned by the ODR stream, and it will live until you call `odr_reset()` on the stream. You cannot use `free(2)` to release the memory. You can decode several data elements (by repeated calls to `odr_setbuf()` and your decoding function), and new memory will be allocated each time. When you do call `odr_reset()`, everything decoded since the last call to `odr_reset()` will be released.

The use of the double indirection can be a little confusing at first (its purpose will become clear later on, hopefully), so an example is in order. We'll encode an integer value, and immediately decode it again using a different stream. A useless, but informative operation.

```
void do_nothing_useful(int value)
{
    ODR encode, decode;
    int *valp, *resvalp;
    char *bufferp;
    int len;

    /* allocate streams */
    if (!(encode = odr_createmem(ODR_ENCODE)))
        return;
    if (!(decode = odr_createmem(ODR_DECODE)))
        return;

    valp = &value;
    if (odr_integer(encode, &valp, 0, 0) == 0)
    {
        printf("encoding went bad\n");
        return;
    }
    bufferp = odr_getbuf(encode, &len);
    printf("length of encoded data is %d\n", len);

    /* now let's decode the thing again */
    odr_setbuf(decode, bufferp, len);
    if (odr_integer(decode, &resvalp, 0, 0) == 0)
    {
        printf("decoding went bad\n");
        return;
    }
    printf("the value is %d\n", *resvalp);

    /* clean up */
    odr_destroy(encode);
    odr_destroy(decode);
}
```

This looks like a lot of work, offhand. In practice, the ODR streams will typically be allocated once, in the beginning of your program (or at the beginning of a new network session), and the encoding and decoding will only take place in a few, isolated places in your program, so the overhead is quite manageable.

Diagnostics

The encoding/decoding functions all return 0 when an error occurs. Until you call `odr_reset()`, you cannot use the stream again, and any function called will immediately return 0.

To provide information to the programmer or administrator, the function

```
void odr_perror(ODR o, char *message);
```

is provided, which prints the `message` argument to `stderr` along with an error message from the stream.

You can also use the function

```
int odr_geterror(ODR o);
```

to get the current error number from the screen. The number will be one of these constants:

Table 9-1. ODR Error codes

code	Description
OMEMORY	Memory allocation failed.
OSYSERR	A system- or library call has failed. The standard diagnostic variable `errno` should be examined to determine the actual error.
OSPACE	No more space for encoding. This will only occur when the user has explicitly provided a buffer for an encoding stream without allowing the system to allocate more space.
OREQUIRED	This is a common protocol error; A required data element was missing during encoding or decoding.
OUNEXPECTED	An unexpected data element was found during decoding.
OOTHER	Other error. This is typically an indication of misuse of the ODR system by the programmer, and also that the diagnostic system isn't as good as it should be, yet.

The character string array

```
char *odr_errlist[]
```

can be indexed by the error code to obtain a human-readable representation of the problem.

Summary and Synopsis

```
#include <odr.h>

ODR odr_createmem(int direction);

void odr_destroy(ODR o);

void odr_reset(ODR o);

char *odr_getbuf(ODR o, int *len);

void odr_setbuf(ODR o, char *buf, int len);

void *odr_malloc(ODR o, int size);

ODR_MEM odr_extract_mem(ODR o);

void odr_release_mem(ODR_MEM r);

int odr_geterror(ODR o);

void odr_perror(char *message);

extern char *odr_errlist[];
```

Programming with ODR

The API of ODR is designed to reflect the structure of ASN.1, rather than BER itself. Future releases may be able to represent data in other external forms.

The interface is based loosely on that of the Sun Microsystems XDR routines. Specifically, each function which corresponds to an ASN.1 primitive type has a dual function. Depending on the settings of the ODR stream which is supplied as a parameter, the function may be used either to encode or decode data. The functions that can be built using these primitive functions, to represent more complex data types, share this quality. The result is that you only have to enter the definition for a type once - and you have the functionality of encoding, decoding (and pretty-printing) all in one unit. The resulting C source code is quite compact, and is a pretty straightforward representation of the source ASN.1 specification.

In many cases, the model of the XDR functions works quite well in this role. In others, it is less elegant. Most of the hassle comes from the optional SEQUENCE members which don’t exist in XDR.

The Primitive ASN.1 Types

ASN.1 defines a number of primitive types (many of which correspond roughly to primitive types in structured programming languages, such as C).

INTEGER

The ODR function for encoding or decoding (or printing) the ASN.1 INTEGER type looks like this:

```
int odr_integer(ODR o, int **p, int optional, const char *name);
```

(we don't allow values that can't be contained in a C integer.)

This form is typical of the primitive ODR functions. They are named after the type of data that they encode or decode. They take an ODR stream, an indirect reference to the type in question, and an `optional` flag (corresponding to the OPTIONAL keyword of ASN.1) as parameters. They all return an integer value of either one or zero. When you use the primitive functions to construct encoders for complex types of your own, you should follow this model as well. This ensures that your new types can be reused as elements in yet more complex types.

The `o` parameter should obviously refer to a properly initialized ODR stream of the right type (encoding/decoding/printing) for the operation that you wish to perform.

When encoding or printing, the function first looks at `* p`. If `* p` (the pointer pointed to by `p`) is a null pointer, this is taken to mean that the data element is absent. If the `optional` parameter is nonzero, the function will return one (signifying success) without any further processing. If the `optional` is zero, an internal error flag is set in the ODR stream, and the function will return 0. No further operations can be carried out on the stream without a call to the function `odr_reset()`.

If `*p` is not a null pointer, it is expected to point to an instance of the data type. The data will be subjected to the encoding rules, and the result will be placed in the buffer held by the ODR stream.

The other ASN.1 primitives have similar functions that operate in similar manners:

BOOLEAN

```
int odr_bool(ODR o, bool_t **p, int optional, const char *name);
```

REAL

Not defined.

NULL

```
int odr_null(ODR o, bool_t **p, int optional, const char *name);
```

In this case, the value of **p is not important. If `*p` is different from the null pointer, the null value is present, otherwise it's absent.

OCTET STRING

```
typedef struct odr_oct
{
    unsigned char *buf;
    int len;
    int size;
} Odr_oct;

int odr_octetstring(ODR o, Odr_oct **p, int optional,
                    const char *name);
```

The `buf` field should point to the character array that holds the octetstring. The `len` field holds the actual length, while the `size` field gives the size of the allocated array (not of interest to you, in most cases). The character array need not be null terminated.

To make things a little easier, an alternative is given for string types that are not expected to contain embedded NULL characters (eg. VisibleString):

```
int odr_cstring(ODR o, char **p, int optional, const char *name);
```

Which encoded or decodes between OCTETSTRING representations and null-terminates C strings.

Functions are provided for the derived string types, eg:

```
int odr_visiblestring(ODR o, char **p, int optional,
                      const char *name);
```

BIT STRING

```
int odr_bitstring(ODR o, Odr_bitmask **p, int optional,
                  const char *name);
```

The opaque type `Odr_bitmask` is only suitable for holding relatively brief bit strings, eg. for options fields, etc. The constant `ODR_BITMASK_SIZE` multiplied by 8 gives the maximum possible number of bits.

A set of macros are provided for manipulating the `Odr_bitmask` type:

```
void ODR_MASK_ZERO(Odr_bitmask *b);

void ODR_MASK_SET(Odr_bitmask *b, int bitno);

void ODR_MASK_CLEAR(Odr_bitmask *b, int bitno);

int ODR_MASK_GET(Odr_bitmask *b, int bitno);
```

The functions are modeled after the manipulation functions that accompany the `fd_set` type used by the `select(2)` call. `ODR_MASK_ZERO` should always be called first on a new bitmask, to initialize the bits to zero.

OBJECT IDENTIFIER

```
int odr_oid(ODR o, Odr_oid **p, int optional, const char *name);
```

The C OID representation is simply an array of integers, terminated by the value -1 (the `Odr_oid` type is synonymous with the `int` type). We suggest that you use the OID database module (see section Object Identifiers) to handle object identifiers in your application.

Tagging Primitive Types

The simplest way of tagging a type is to use the `odr_implicit_tag()` or `odr_explicit_tag()` macros:

```
int odr_implicit_tag(ODR o, Odr_fun fun, int class, int tag,
                     int optional, const char *name);

int odr_explicit_tag(ODR o, Odr_fun fun, int class, int tag,
                     int optional, const char *name);
```

To create a type derived from the integer type by implicit tagging, you might write:

```
     MyInt ::= [210] IMPLICIT INTEGER
```

In the ODR system, this would be written like:

```
int myInt(ODR o, int **p, int optional, const char *name)
{
    return odr_implicit_tag(o, odr_integer, p,
    ODR_CONTEXT, 210, optional, name);
}
```

The function `myInt()` can then be used like any of the primitive functions provided by ODR. Note that the behavior of `odr_explicit()` and `odr_implicit()` macros act exactly the same as the functions they are applied to - they respond to error conditions, etc, in the same manner - they simply have three extra parameters. The class parameter may take one of the values: `ODR_CONTEXT`, `ODR_PRIVATE`, `ODR_UNIVERSAL`, or `/ODR_APPLICATION`.

Constructed Types

Constructed types are created by combining primitive types. The ODR system only implements the SEQUENCE and SEQUENCE OF constructions (although adding the rest of the container types should be simple enough, if the need arises).

For implementing SEQUENCEs, the functions

```
int odr_sequence_begin(ODR o, void *p, int size, const char *name);
int odr_sequence_end(ODR o);
```

are provided.

The `odr_sequence_begin()` function should be called in the beginning of a function that implements a SEQUENCE type. Its parameters are the ODR stream, a pointer (to a pointer to the type you're implementing), and the `size` of the type (typically a C structure). On encoding, it returns 1 if `* p` is a null pointer. The `size` parameter is ignored. On decoding, it returns 1 if the type is found in the data stream. `size` bytes of memory are allocated, and `*p` is set to point to this space. `odr_sequence_end()` is called at the end of the complex function. Assume that a type is defined like this:

```
MySequence ::= SEQUENCE {
     intval INTEGER,
     boolval BOOLEAN OPTIONAL
}
```

The corresponding ODR encoder/decoder function and the associated data structures could be written like this:

```
typedef struct MySequence
{
    int *intval;
    bool_t *boolval;
} MySequence;

int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_sequence_begin(o, p, sizeof(**p), name) == 0)
        return optional && odr_ok(o);
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o);
}
```

Note the 1 in the call to `odr_bool()`, to mark that the sequence member is optional. If either of the member types had been tagged, the macros `odr_implicit()` or `odr_explicit()` could have been used. The new function can be used exactly like the standard functions provided with ODR. It will encode, decode or pretty-print a data value of the `MySequence` type. We like to name types with an initial capital, as done in ASN.1 definitions, and to name the corresponding function with the first

character of the name in lower case. You could, of course, name your structures, types, and functions any way you please - as long as you're consistent, and your code is easily readable. `odr_ok` is just that - a predicate that returns the state of the stream. It is used to ensure that the behavior of the new type is compatible with the interface of the primitive types.

Tagging Constructed Types

> **Note:** See section Tagging Primitive types for information on how to tag the primitive types, as well as types that are already defined.

Implicit Tagging

Assume the type above had been defined as

```
MySequence ::= [10] IMPLICIT SEQUENCE {
      intval INTEGER,
      boolval BOOLEAN OPTIONAL
}
```

You would implement this in ODR by calling the function

```
int odr_implicit_settag(ODR o, int class, int tag);
```

which overrides the tag of the type immediately following it. The macro `odr_implicit()` works by calling `odr_implicit_settag()` immediately before calling the function pointer argument. Your type function could look like this:

```
int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_implicit_settag(o, ODR_CONTEXT, 10) == 0 ||
        odr_sequence_begin(o, p, sizeof(**p), name) == 0)
        return optional && odr_ok(o);
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o);
}
```

The definition of the structure `MySequence` would be the same.

Explicit Tagging

Explicit tagging of constructed types is a little more complicated, since you are in effect adding a level of construction to the data.

Assume the definition:

```
MySequence ::= [10] IMPLICIT SEQUENCE {
   intval INTEGER,
   boolval BOOLEAN OPTIONAL
}
```

Since the new type has an extra level of construction, two new functions are needed to encapsulate the base type:

```
int odr_constructed_begin(ODR o, void *p, int class, int tag,
                          const char *name);

int odr_constructed_end(ODR o);
```

Assume that the IMPLICIT in the type definition above were replaced with EXPLICIT (or that the IMPLICIT keyword were simply deleted, which would be equivalent). The structure definition would look the same, but the function would look like this:

```
int mySequence(ODR o, MySequence **p, int optional, const char *name)
{
    if (odr_constructed_begin(o, p, ODR_CONTEXT, 10, name) == 0)
        return optional && odr_ok(o);
    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p));
    if (odr_sequence_begin(o, p, sizeof(**p), 0) == 0)
    {
        *p = 0; /* this is almost certainly a protocol error */
        return 0;
    }
    return
        odr_integer(o, &(*p)->intval, 0, "intval") &&
        odr_bool(o, &(*p)->boolval, 1, "boolval") &&
        odr_sequence_end(o) &&
        odr_constructed_end(o);
}
```

Notice that the interface here gets kind of nasty. The reason is simple: Explicitly tagged, constructed types are fairly rare in the protocols that we care about, so the esthetic annoyance (not to mention the dangers of a cluttered interface) is less than the time that would be required to develop a better interface. Nevertheless, it is far from satisfying, and it's a point that will be worked on in the future. One option for you would be to simply apply the `odr_explicit()` macro to the first function, and not have to worry about `odr_constructed_*` yourself. Incidentally, as you might have guessed, the `odr_sequence_` functions are themselves implemented using the `/odr_constructed_` functions.

SEQUENCE OF

To handle sequences (arrays) of a specific type, the function

```
int odr_sequence_of(ODR o, int (*fun)(ODR o, void *p, int optional),
```

```
                    void *p, int *num, const char *name);
```

The `fun` parameter is a pointer to the decoder/encoder function of the type. `p` is a pointer to an array of pointers to your type. `num` is the number of elements in the array.

Assume a type

```
MyArray ::= SEQUENCE OF INTEGER
```

The C representation might be

```
typedef struct MyArray
{
    int num_elements;
    int **elements;
} MyArray;
```

And the function might look like

```
int myArray(ODR o, MyArray **p, int optional, const char *name)
{
    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p));
    if (odr_sequence_of(o, odr_integer, &(*p)->elements,
        &(*p)->num_elements, name))
        return 1;
    *p = 0;
        return optional && odr_ok(o);
}
```

CHOICE Types

The choice type is used fairly often in some ASN.1 definitions, so some work has gone into streamlining its interface.

CHOICE types are handled by the function:

```
int odr_choice(ODR o, Odr_arm arm[], void *p, void *whichp,
               const char *name);
```

The `arm` array is used to describe each of the possible types that the CHOICE type may assume. Internally in your application, the CHOICE type is represented as a discriminated union. That is, a C union accompanied by an integer (or enum) identifying the active 'arm' of the union. `whichp` is a pointer to the union discriminator. When encoding, it is examined to determine the current type. When decoding, it is set to reference the type that was found in the input stream.

The Odr_arm type is defined thus:

```
typedef struct odr_arm
{
    int tagmode;
    int class;
    int tag;
    int which;
    Odr_fun fun;
    char *name;
} Odr_arm;
```

The interpretation of the fields are:

tagmode

Either `ODR_IMPLICIT`, `ODR_EXPLICIT`, or `ODR_NONE` (-1) to mark no tagging.

which

The value of the discriminator that corresponds to this CHOICE element. Typically, it will be a #defined constant, or an enum member.

fun

A pointer to a function that implements the type of the CHOICE member. It may be either a standard ODR type or a type defined by yourself.

name

Name of tag.

A handy way to prepare the array for use by the `odr_choice()` function is to define it as a static, initialized array in the beginning of your decoding/encoding function. Assume the type definition:

```
MyChoice ::= CHOICE {
    untagged INTEGER,
    tagged   [99] IMPLICIT INTEGER,
    other    BOOLEAN
}
```

Your C type might look like

```
typedef struct MyChoice
{
    enum
    {
        MyChoice_untagged,
        MyChoice_tagged,
        MyChoice_other
    } which;
    union
    {
        int *untagged;
        int *tagged;
```

```
        bool_t *other;
    } u;
};
```

And your function could look like this:

```
int myChoice(ODR o, MyChoice **p, int optional, const char *name)
{
    static Odr_arm arm[] =
    {
      {-1, -1, -1, MyChoice_untagged, odr_integer, "untagged"},
      {ODR_IMPLICIT, ODR_CONTEXT, 99, MyChoice_tagged, odr_integer,
      "tagged"},
      {-1, -1, -1, MyChoice_other, odr_boolean, "other"},
      {-1, -1, -1, -1, 0}
    };

    if (o->direction == ODR_DECODE)
        *p = odr_malloc(o, sizeof(**p);
    else if (!*p)
        return optional && odr_ok(o);

    if (odr_choice(o, arm, &(*p)->u, &(*p)->which), name)
        return 1;
    *p = 0;
        return optional && odr_ok(o);
}
```

In some cases (say, a non-optional choice which is a member of a sequence), you can "embed" the union and its discriminator in the structure belonging to the enclosing type, and you won’t need to fiddle with memory allocation to create a separate structure to wrap the discriminator and union.

The corresponding function is somewhat nicer in the Sun XDR interface. Most of the complexity of this interface comes from the possibility of declaring sequence elements (including CHOICEs) optional.

The ASN.1 specifications naturally requires that each member of a CHOICE have a distinct tag, so they can be told apart on decoding. Sometimes it can be useful to define a CHOICE that has multiple types that share the same tag. You’ll need some other mechanism, perhaps keyed to the context of the CHOICE type. In effect, we would like to introduce a level of context-sensitiveness to our ASN.1 specification. When encoding an internal representation, we have no problem, as long as each CHOICE member has a distinct discriminator value. For decoding, we need a way to tell the choice function to look for a specific arm of the table. The function

```
void odr_choice_bias(ODR o, int what);
```

provides this functionality. When called, it leaves a notice for the next call to `odr_choice()` to be called on the decoding stream `o` that only the `arm` entry with a `which` field equal to `what` should be tried.

The most important application (perhaps the only one, really) is in the definition of application-specific EXTERNAL encoders/decoders which will automatically decode an ANY member given the direct or indirect reference.

Debugging

The protocol modules are suffering somewhat from a lack of diagnostic tools at the moment. Specifically ways to pretty-print PDUs that aren't recognized by the system. We'll include something to this end in a not-too-distant release. In the meantime, what we do when we get packages we don't understand is to compile the ODR module with `ODR_DEBUG` defined. This causes the module to dump tracing information as it processes data units. With this output and the protocol specification (Z39.50), it is generally fairly easy to see what goes wrong.

Chapter 10. The COMSTACK Module

Synopsis (blocking mode)

```
COMSTACK stack;
char *buf = 0;
int size = 0, length_incoming;
char *protocol_package;
int protocol_package_length;
char server_address_str[] = "myserver.com:2100";
void *server_address_ip;
int status;

stack = cs_create(tcpip_type, 1, PROTO_Z3950);
if (!stack) {
    perror("cs_create");  /* use perror() here since we have no stack yet */
    exit(1);
}

server_address_ip = cs_addrstr (stack, server_address_str);

status = cs_connect(stack, server_address_ip);
if (status != 0) {
    cs_perror(stack, "cs_connect");
    exit(1);
}

status = cs_put(stack, protocol_package, protocol_package_length);
if (status) {
    cs_perror(stack, "cs_put");
    exit(1);
}

/* Now get a response */

length_incoming = cs_get(stack, &buf, &size);
if (!length_incoming) {
    fprintf(stderr, "Connection closed\n");
    exit(1);
} else if (length_incoming < 0) {
    cs_perror(stack, "cs_get");
    exit(1);
}

/* Do stuff with buf here */

/* clean up */
cs_close(stack);
if (buf)
    free(buf);
```

Introduction

The COMSTACK subsystem provides a transparent interface to different types of transport stacks for the exchange of BER-encoded data and HTTP packets. At present, the RFC1729 method (BER over TCP/IP), local UNIX socket and an experimental SSL stack are supported, but others may be added in time. The philosophy of the module is to provide a simple interface by hiding unused options and facilities of the underlying libraries. This is always done at the risk of losing generality, and it may prove that the interface will need extension later on.

> **Note:** There hasn't been interest in the XTImOSI stack for some years. Therefore, it is no longer supported.

The interface is implemented in such a fashion that only the sub-layers constructed to the transport methods that you wish to use in your application are linked in.

You will note that even though simplicity was a goal in the design, the interface is still orders of magnitudes more complex than the transport systems found in many other packages. One reason is that the interface needs to support the somewhat different requirements of the different lower-layer communications stacks; another important reason is that the interface seeks to provide a more or less industrial-strength approach to asynchronous event-handling. When no function is allowed to block, things get more complex - particularly on the server side. We urge you to have a look at the demonstration client and server provided with the package. They are meant to be easily readable and instructive, while still being at least moderately useful.

Common Functions

Managing Endpoints

```
COMSTACK cs_create(CS_TYPE type, int blocking, int protocol);
```

Creates an instance of the protocol stack - a communications endpoint. The `type` parameter determines the mode of communication. At present the following values are supported:

`tcpip_type`

TCP/IP (BER over TCP/IP or HTTP over TCP/IP)

`ssl_type`

Secure Socket Layer (SSL). This COMSTACK is experimental and is not fully implemented. If HTTP is used, this effectively is HTTPS.

`unix_type`

Unix socket (unix only). Local Transfer via file socket. See unix(7).

The `cs_create` function returns a null-pointer if a system error occurs. The `blocking` parameter should be one if you wish the association to operate in blocking mode, zero otherwise. The `protocol` field should be `PROTO_Z3950` or `PROTO_HTTP`. Protocol `PROTO_SR` is no longer supported.

```
int cs_close(COMSTACK handle);
```

Closes the connection (as elegantly as the lower layers will permit), and releases the resources pointed to by the `handle` parameter. The `handle` should not be referenced again after this call.

Note: We really need a soft disconnect, don't we?

Data Exchange

```
int cs_put(COMSTACK handle, char *buf, int len);
```

Sends `buf` down the wire. In blocking mode, this function will return only when a full buffer has been written, or an error has occurred. In nonblocking mode, it's possible that the function will be unable to send the full buffer at once, which will be indicated by a return value of 1. The function will keep track of the number of octets already written; you should call it repeatedly with the same values of `buf` and `len`, until the buffer has been transmitted. When a full buffer has been sent, the function will return 0 for success. -1 indicates an error condition (see below).

```
int cs_get(COMSTACK handle, char **buf, int *size);
```

Receives a PDU or HTTP Response from the peer. Returns the number of bytes read. In nonblocking mode, it is possible that not all of the packet can be read at once. In this case, the function returns 1. To simplify the interface, the function is responsible for managing the size of the buffer. It will be reallocated if necessary to contain large packages, and will sometimes be moved around internally by the subsystem when partial packages are read. Before calling `cs_get` for the fist time, the buffer can be initialized to the null pointer, and the length should also be set to 0 - cs_get will perform a `malloc(2)` on the buffer for you. When a full buffer has been read, the size of the package is returned (which will always be greater than 1). -1 indicates an error condition.

See also the `cs_more()` function below.

```
int cs_more(COMSTACK handle);
```

The `cs_more()` function should be used in conjunction with `cs_get` and `select(2)`. The `cs_get()` function will sometimes (notably in the TCP/IP mode) read more than a single protocol package off the network. When this happens, the extra package is stored by the subsystem. After calling `cs_get()`, and before waiting for more input, You should always call `cs_more()` to check if there's a full protocol

package already read. If `cs_more()` returns 1, `cs_get()` can be used to immediately fetch the new package. For the mOSI subsystem, the function should always return 0, but if you want your stuff to be protocol independent, you should use it.

> **Note:** The `cs_more()` function is required because the RFC1729-method does not provide a way of separating individual PDUs, short of partially decoding the BER. Some other implementations will carefully nibble at the packet by calling `read(2)` several times. This was felt to be too inefficient (or at least clumsy) - hence the call for this extra function.

```
int cs_look(COMSTACK handle);
```

This function is useful when you're operating in nonblocking mode. Call it when `select(2)` tells you there's something happening on the line. It returns one of the following values:

CS_NONE

No event is pending. The data found on the line was not a complete package.

CS_CONNECT

A response to your connect request has been received. Call `cs_rcvconnect` to process the event and to finalize the connection establishment.

CS_DISCON

The other side has closed the connection (or maybe sent a disconnect request - but do we care? Maybe later). Call `cs_close` to close your end of the association as well.

CS_LISTEN

A connect request has been received. Call `cs_listen` to process the event.

CS_DATA

There's data to be found on the line. Call `cs_get` to get it.

> **Note:** You should be aware that even if `cs_look()` tells you that there's an event event pending, the corresponding function may still return and tell you there was nothing to be found. This means that only part of a package was available for reading. The same event will show up again, when more data has arrived.

```
int cs_fileno(COMSTACK h);
```

Returns the file descriptor of the association. Use this when file-level operations on the endpoint are required (`select(2)` operations, specifically).

Client Side

```
int cs_connect(COMSTACK handle, void *address);
```

Initiate a connection with the target at `address` (more on addresses below). The function will return 0 on success, and 1 if the operation does not complete immediately (this will only happen on a nonblocking endpoint). In this case, use `cs_rcvconnect` to complete the operation, when `select(2)` or `poll(2)` reports input pending on the association.

```
int cs_rcvconnect(COMSTACK handle);
```

Complete a connect operation initiated by `cs_connect()`. It will return 0 on success; 1 if the operation has not yet completed (in this case, call the function again later); -1 if an error has occurred.

Server Side

To establish a server under the inetd server, you can use

```
COMSTACK cs_createbysocket(int socket, CS_TYPE type, int blocking,
                           int protocol);
```

The `socket` parameter is an established socket (when your application is invoked from inetd, the socket will typically be 0. The following parameters are identical to the ones for `cs_create`.

```
int cs_bind(COMSTACK handle, void *address, int mode)
```

Binds a local address to the endpoint. Read about addresses below. The `mode` parameter should be either `CS_CLIENT` or `CS_SERVER`.

```
int cs_listen(COMSTACK handle, char *addr, int *addrlen);
```

Call this to process incoming events on an endpoint that has been bound in listening mode. It will return 0 to indicate that the connect request has been received, 1 to signal a partial reception, and -1 to indicate an error condition.

```
COMSTACK cs_accept(COMSTACK handle);
```

This finalizes the server-side association establishment, after cs_listen has completed successfully. It returns a new connection endpoint, which represents the new association. The application will typically wish to fork off a process to handle the association at this point, and continue listen for new connections on the old `handle`.

You can use the call

```
char *cs_addrstr(COMSTACK);
```

on an established connection to retrieve the host-name of the remote host.

> **Note:** You may need to use this function with some care if your name server service is slow or unreliable

Addresses

The low-level format of the addresses are different depending on the mode of communication you have chosen. A function is provided by each of the lower layers to map a user-friendly string-form address to the binary form required by the lower layers.

```
void *cs_straddr(COMSTACK handle, const char *str);
```

The format for TCP/IP and SSL addresses is:

```
<host> [ ':' <portnum> ]
```

The `hostname` can be either a domain name or an IP address. The port number, if omitted, defaults to 210.

For TCP/IP and SSL transport modes, the special hostname "@" is mapped to any local address (the manifest constant `INADDR_ANY`). It is used to establish local listening endpoints in the server role.

For UNIX sockets, the format of an address is the socket filename.

When a connection has been established, you can use

```
char *cs_addrstr(COMSTACK h);
```

to retrieve the host name of the peer system. The function returns a pointer to a static area, which is overwritten on the next call to the function.

A fairly recent addition to the COMSTACK module is the utility function

```
COMSTACK cs_create_host (const char *str, int blocking, void **vp);
```

which is just a wrapper for `cs_create` and `cs_straddr`. The *`str`* is similar to that described for `cs_straddr` but with a prefix denoting the COMSTACK type. Prefixes supported are `tcp:`, `unix:` and `ssl:` for TCP/IP, UNIX and SSL respectively. If no prefix is given, then TCP/IP is used. The *`blocking`* is passed to function `cs_create`. The third parameter *`vp`* is a pointer to COMSTACK stack type specific values. For SSL (ssl_type) *`vp`* is an already create OpenSSL CTX. For TCP/IP and UNIX *`vp`* is unused (can be set to `NULL`.

Diagnostics

All functions return -1 if an error occurs. Typically, the functions will return 0 on success, but the data exchange functions (`cs_get`, `cs_put`, `cs_more`) follow special rules. Consult their descriptions.

When a function (including the data exchange functions) reports an error condition, use the function `cs_errno()` to determine the cause of the problem. The function

```
void cs_perror(COMSTACK handle char *message);
```

works like `perror(2)` and prints the `message` argument, along with a system message, to `stderr`. Use the character array

```
extern const char *cs_errlist[];
```

to get hold of the message, if you want to process it differently. The function

```
const char *cs_stackerr(COMSTACK handle);
```

Returns an error message from the lower layer, if one has been provided.

Summary and Synopsis

```
#include <yaz/comstack.h>

#include <yaz/tcpip.h>  /* this is for TCP/IP and SSL support */
#include <yaz/unix.h>   /* this is for UNIX sockeL support */


COMSTACK cs_create(CS_TYPE type, int blocking, int protocol);

COMSTACK cs_createbysocket(int s, CS_TYPE type, int blocking,
                           int protocol);
COMSTACK cs_create_host (const char *str, int blocking,
                         void **vp);

int cs_bind(COMSTACK handle, int mode);

int cs_connect(COMSTACK handle, void *address);

int cs_rcvconnect(COMSTACK handle);

int cs_listen(COMSTACK handle);

COMSTACK cs_accept(COMSTACK handle);

int cs_put(COMSTACK handle, char *buf, int len);

int cs_get(COMSTACK handle, char **buf, int *size);
```

```
int cs_more(COMSTACK handle);

int cs_close(COMSTACK handle);

int cs_look(COMSTACK handle);

void *cs_straddr(COMSTACK handle, const char *str);

char *cs_addrstr(COMSTACK h);

extern int cs_errno;

void cs_perror(COMSTACK handle char *message);

const char *cs_stackerr(COMSTACK handle);

extern const char *cs_errlist[];
```

Chapter 11. Future Directions

We have a new and better version of the front-end server on the drawing board. Resources and external commitments will govern when we'll be able to do something real with it. Features should include greater flexibility, greater support for access/resource control, and easy support for Explain (possibly with Zebra as an extra database engine).

YAZ is a BER toolkit and as such should support all protocols out there based on that. We'd like to see running ILL applications. It shouldn't be that hard. Another thing that would be interesting is LDAP. Maybe a generic framework for doing IR using both LDAP and Z39.50 transparently.

The SOAP implementation is incomplete. In the future we hope to add more features to it. Perhaps make a WSDL/XML Schema compiler. The authors of libxml2 are already working on XML Schema / RelagNG compilers so this may not be too hard.

It would be neat to have a proper module mechanism for the Generic Frontend Server so that backend would be dynamically loaded (as shared objects / DLLs).

Other than that, YAZ generally moves in the directions which appear to make the most people happy (including ourselves, as prime users of the software). If there's something you'd like to see in here, then drop us a note and let's see what we can come up with.

Appendix A. License

Index Data Copyright

Copyright © 1995-2003 Index Data ApS.

Permission to use, copy, modify, distribute, and sell this software and its documentation, in whole or in part, for any purpose, is hereby granted, provided that:

1. This copyright and permission notice appear in all copies of the software and its documentation. Notices of copyright or attribution which appear at the beginning of any file must remain unchanged.

2. The names of Index Data or the individual authors may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED, OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL INDEX DATA BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Additional Copyright Statements

The optional CCL query language interpreter is covered by the following license:

Copyright © 1995, the EUROPAGATE consortium (see below).

The EUROPAGATE consortium members are:
University College Dublin
Danmarks Teknologiske Videnscenter
An Chomhairle Leabharlanna
Consejo Superior de Investigaciones Cientificas

Permission to use, copy, modify, distribute, and sell this software and its documentation, in whole or in part, for any purpose, is hereby granted, provided that:

1. This copyright and permission notice appear in all copies of the software and its documentation. Notices of copyright or attribution which appear at the beginning of any file must remain unchanged.

2. The names of EUROPAGATE or the project partners may not be used to endorse or promote products derived from this software without specific prior written permission.

3. Users of this software (implementors and gateway operators) agree to inform the EUROPAGATE consortium of their use of the software. This information will be used to evaluate the EUROPAGATE project and the software, and to plan further developments. The consortium may use the information in later publications.

4. Users of this software agree to make their best efforts, when documenting their use of the software, to acknowledge the EUROPAGATE consortium, and the role played by the software in their work.

THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT WARRANTY OF ANY KIND, EXPRESS, IMPLIED, OR OTHERWISE, INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. IN NO EVENT SHALL THE EUROPAGATE CONSORTIUM OR ITS MEMBERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER OR NOT ADVISED OF THE POSSIBILITY OF DAMAGE, AND ON ANY THEORY OF LIABILITY, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Appendix B. About Index Data

Index Data is a consulting and software-development enterprise that specializes in library and information management systems. Our interests and expertise span a broad range of related fields, and one of our primary, long-term objectives is the development of a powerful information management system with open network interfaces and hyper-media capabilities.

We make this software available free of charge, on a fairly unrestrictive license; as a service to the networking community, and to further the development of quality software for open network communication.

We'll be happy to answer questions about the software, and about ourselves in general.

Index Data Aps
Købmagergade 43
1150 Copenhagen K
Denmark
Phone +45 3341 0100
Fax +45 3341 0101
Email <info@indexdata.dk>

The Hacker's Jargon File has the following to say about the use of the prefix "YA" in the name of a software product.

[Yet Another. adj. 1. Of your own work: A humorous allusion often used in titles to acknowledge that the topic is not original, though the content is. As in "Yet Another AI Group" or "Yet Another Simulated Annealing Algorithm". 2. Of others' work: Describes something of which there are already far too many.]

Appendix C. Credits

This appendix lists individuals that have contributed in the development of YAZ. Some have contributed with code, while others have provided bug fixes or suggestions. If we're missing somebody, of if you, for whatever reason, don't like to be listed here, let us know.

- Dimitrios Andreadis
- Morten Bøgeskov
- Rocco Carbone
- Matthew Carey
- Irina Dijour
- Hans van Dalen
- Hans van den Dool
- Franck Falcoz
- Kevin Gamiel
- Morten Garkier Hendriksen
- Morten Holmqvist
- Ian Ibbotson
- Shigeru Ishida
- David Johnson
- Oleg Kolobov
- Kang-Jin Lee
- Stefan Lohrum
- Ronald van der Meer
- Thomas W. Place
- Jacob Chr. Poulsen
- Ko van der Sloot
- Mike Taylor
- Rustam T. Usmanov
- Charles Woodfield
- Tom André Øverland